ALCOR SYSTEMS
800 W. Garland Ave.
Garland,Texas 75040

SOFTWARE LICENSE AGREEMENT

Please read this license agreement carefully.

All Alcor Systems programs are sold on the condition that the purchaser agrees to the following license. If you do not agree to the terms contained in this license, return the packaged diskette(s) UNOPENED to the place of purchase, and the purchase price will be refunded. If you agree to the terms contained in this license, fill out the registration information and license agreement.

Alcor Systems may discontinue any license or terminate this Agreement if the Customer fails to comply with any of the terms and conditions of this Agreement.

Alcor Systems hereby grants the undersigned customer a single COMPUTER SYSTEM LICENSE which is non-transferable. A separate license is required for each additional COMPUTER SYSTEM it is to be used with, except herein described. All supplied materials are subject to this LICENSE restriction including any printed material. Copies of printed material must be obtained from Alcor Systems.

BACKUP

Any Licensed Programs supplied by Alcor Systems under this agreement may be copied by the undersigned customer for backup or archival purposes in sufficient number for customer's use of the licensed programs.

The Customer agrees to maintain appropriate records of the number and location of all such copies of Licensed Programs.

The Customer agrees to reproduce and include the copyright notice of Alcor Systems on all copies, in whole or in part, in any form, including partial copies or modifications, of Licensed Programs made hereunder.

The Customer agrees not to provide or otherwise make available any Licensed Program, including but not limited to, program listings, object code and source code, in any form, to any person other than customer's or Alcor Systems employees.

DISCLAIMER OF WARRANTY:

Alcor Systems makes no warranties with respect to the Licensed Programs. The sole obligation of Alcor Systems shall be to make available all published modifications or patches made by Alcor Systems to Licensed Programs which are published within one (1) year from date of purchase, provided the Customer has returned the Registration card, and signed license agreement.

LIMITATION OF LIABILITY:
THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL ALCOR SYSTEMS BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF ALCOR SYSTEMS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are to that extent to be deemed omitted.

The intent of this Agreement is not to restrict in any way the normal use of Alcor Systems software, but is simply to protect Alcor Systems from any fraudulent use or mass reproduction and piracy of it's software. This is a standard commercial software agreement and gives more latitude than many such agreements. It's sole purpose is to make the Customer responsible for the protection of Licensed Programs. Alcor Systems feels that this approach is better than copy protection schemes currently used by some companies. Any violations of this agreement may be grounds for Alcor Systems to take appropriate legal action against said Customer.

ADDENDUM: PROGRAMS PRODUCED WITH THE PASCAL LANGUAGE SYSTEM

The Customer may produce object programs that include Alcor Systems runtime support and may distribute them as desired. The runtime support referred to is the Pascal interpreter, and any Alcor Systems system interface modules that are normally linked to the translated program module with the Pascal linking loader or linkage editor. The only requirement is that the documentation supplied by the Customer with distributed programs that include Alcor Systems runtime support must specifically include a notice that the program includes Alcor Systems runtime support modules from the Alcor Systems Pascal Language System.

I hereby the undersigned, understand and agree to the above licensing agreement.

Signature

Date

ALCOR PASCAL REGISTRATION CARD

SERIAL NUMBER ______________________

Name ______________________

Address ______________________

City,State,Zip ______________________

Place of purchase ______________________

Date ______________________

ALCOR PASCAL SERVICE POLICY

PATCHES

Patches are supplied by two different methods.

(1) Alcor Systems will send printed patches to be applied by the customer using the Alcor Pascal Patch program. All patches must be applied by this method to ensure system integrity simply due to the system size and complexity.

(2) Alcor Systems will optionally send patches on diskette media for use with the Alcor patch program. The diskettes may be obtained directly from Alcor Systems for a nominal charge. See the Alcor Pascal Newsletter for current information.

VERSION UPGRADES

Any new Alcor Pascal release may be obtained by registered customers for a version upgrade fee which is set at the time of the release.

********************* IMPORTANT - READ THIS FIRST ************************

Upon receipt of the Alcor Systems, ALCOR PASCAL please check the contents of the supplied material for the following items: Beginners manual, Editor manual, System manual, Pascal Tutorial manual, Language Reference manual, two/three 5 -1/4 " diskettes with a TRS80 model I / III and Quick Reference programming card.

Please observe the following steps carefully !

1) Read the license agreement and sign it. Fill out the registration slip and return with the signed license agreement. This is an important step. If the registration and license agreement is not received within 30 days of purchase, your update services and warranty may be void.

2) Remove the supplied diskettes and make a back-up immediately ! Use the BACKUP technique that is applicable to your TRSDOS or CP/M compatible operating system. The supplied formats are: TRSDOS Model I - 35 track, single density. Model III - 40 track double density. CPM - 77 track single density, single sided. All supplied 5 1/4 diskettes come write protected with a small adhesive tab. Do not remove this tab. All 8 " supplied diskettes are write protected without the adhesive tab. Label the backup diskettes as the master Alcor Pascal diskettes. Never use the original diskettes for normal use. This will prevent a hardware or software failure from causing disaster.

3) Once diskette back-ups have been made, you may wish to reconfigure additional diskettes by moving the various files around. This may be particularly true on a TRS80. It has been found that a convenient TRS80 configuration is to have the Blaise text editor on the operating system diskette. Explanations of the supplied files may be found in the Beginners guide. Additional information about the Blaise editing systems files may be found in the Blaise editor manual. There are no copy or locking systems on the Alcor Pascal system files. They may be freely copied from diskette to diskette.

SUGGESTIONS ON HOW TO USE THE MANUALS

The supplied documentation set was designed to aid new Pascal programmers. It is not simply a technical reference manual as many language system vendors supply. A common practice in the industry is to simply supply a Jensen and Wirth reference book and let beginners beware. First read the Beginners section ! It describes how to invoke the Blaise text editor and enter a simple Pascal program, translate it and execute it. It is NOT a detailed explanation of all aspects of the Alcor Pascal system. A next logical step would be to read the Blaise text editor manual and try some of its advanced features. For experienced Pascal programmers, the System Implementation manual, and the Language Reference manual will provide all the help necessary to write any Pascal program. If you are a novice, it is suggested that you read the Pascal Tutorial and try entering and executing the example programs. The Tutorial's example programs are supplied on the diskette in source form. You may wish to compile and execute them. A Master Cross Reference Index is supplied with the documentation.

Pascal 1.2A Release Notes

Overview

The differences between Pascal 1.2 and 1.2A are primarily internal patches to correct reported bugs. There are enhancements to to the text editor which allow printable characters not on the TRS80 keyboard to be generated with clear key sequences. Pascal 1.2A has all the known bugs at this time corrected. Registered Pascal 1.2 owners may apply patches to version 1.2 with the supplied Alcor patch program.

Since this is not a major version upgrade, patches will be be published in the first Alcor newsletter available sometime in June 1982, or you may send $ 8.00 + shipping to Alcor Systems for a disk containing the published patches for use in conjunction with the Alcor Patch program. All REGISTERED owners will receive the Alcor Newsletter. In printed form, the patches are over 4 pages long.

************** NOTICE TO PASCAL 1.2A OWNERS **************

The release disks have already been corrected and require no patching of any kind to correct bugs. The only patching necessary is for Pascal operation under the different operating systems. See the supplied Patch instruction sheet.

Known Bugs in Pascal 1.2

PASCAL COMPILER

A. Sets

The IN operator sometimes returns true when <item> is not a member of the set when the ordinal of the <item> is greater than the ordinal of the highest member actually present in the <set expression>.

B. Procedure calls

When making procedure calls of the form:

```
IF <expression> THEN <procedure call> else <statement>
```

the compiler will incorrectly flag a syntax error if the procedure has no parameters.

(c) Copyright 1982 Alcor Systems

C. Hexadecimal constants

If hexadecimal constants are declared greater than #8000 <hex 8000> the compiler incorrectly stores the value.

D. File of Char

When a FILE OF CHAR is declared, and the generated file contains the control characters CR <#0D>, LF <#0A>, TAB <#09> and Control Z <#1A>, they will be treated like they are in a text file and cause special processing when the file is read by Pascal. The correction causes them to be treated like any other character.

Known Bugs in Release 1.2

EDITOR

A. "RENAME FAILED CRASH"

Under certain conditions that are highly operating system and data file dependent, upon exiting from the editor, a "RENAME FAILED" error flag will be displayed. This occasionally results in a file loss. This is caused by an improper filename termination in the file descriptor <no CR termination of filename string> by the editor runtime.

Enhancements in Release 1.2A

EDITOR

A. I/O error detection and recovery

An enhancement to allow for better I/O error correction and recovery has been made in the text editor. When a disk error has been detected, the message:

IO ERROR

followed by the operating system error code is displayed. You may type "Q" to allow the error to pass or any other key to cause a retry.

(c) Copyright 1982 Alcor Systems

B. Screen update change

In 1.2 when the cursor is at the bottom of the display and the enter key is hit, the screen is completely cleared and rewritten. Release 1.2A causes the screen to scroll by sending a line feed to the screen driver. This increases the effective scroll rate.

C. Character generation

In 1.2A certain characters not available on the TRS80 keyboard may be generated by using the clear key sequences. They are:

Sequence		character
clear 1	-------	[
clear 2	-------	]
clear 3	-------	^
clear 4	-------	{
clear 5	-------	}
clear 6	-------	|
clear 7	-------	/

D. Clear key definition

An optional patch is available to allow the clear key to be redefined to the: Shift Down Arrow (Model I only)
or to the: / key on the Model I or Model III.

(c) Copyright 1982 Alcor Systems

Special Notice about the Alcor Pascal Patch program

IT IS RECOMMENDED THAT ALL PATCHING BE DONE UNDER TRSDOS

The Alcor Patch program may be used to apply any patches to the system while operating under TRSDOS on the model I or III. This includes patching the system with the supplied conversion patches which allows the system to execute under the various supported operating systems such as LDOS, NEWDOS and DOSPLUS. (See the supplied patch program instructions for specific information about your computer and operating system combination) After patching has been performed under the TRSDOS operating system, the Pascal system may then be copied to any of the ALCOR supported operating systems using the normal operating system utilities.

IF YOU MUST PATCH WHILE EXECUTING UNDER ANY OF THE ALCOR SUPPORTED OPERATING SYSTEMS OTHER THAN TRSDOS

Generally, if the Pascal system must be patched for execution under your computer and operating system combination, then the PATCH program should be patched first. This particular patch for the PATCH program may be applied while under ANY of the Alcor supported operating systems.
The following patch may be entered into a text file with any legal filename. IE; SPECIAL/PAT You may use the text editor if creating this file under TRSDOS and copy it to the desired operating system.

For Model I systems

```
F, PATCH/CMD, ALCOR
P,0FA1,054E,0001,13,00
P,0B16,0529,0001,00,03
W,F589
E
```

For Model III systems

```
F, PATCH/CMD, ALCOR
P,0FA1,054E,0001,00,13
P,0B16,0529,0001,03,00
W,F589
E
```

(c) Copyright 1982 Alcor Systems

PATCH PROGRAM

The initial release of ALCOR PASCAL requires patching to execute correctly on some operating systems. This is due to incompatibilities among the various TRS80 operating systems. On the TRS80 model I, patches are applied for NEWDOS80 and DOSPLUS. On the TRS80 model III, patches are applied for LDOS.

```
*********************************************
*                  WARNING                  *
*  NEVER apply patches to the original      *
*        release disk.  Make a copy and     *
*        apply patches to the copy.         *
*********************************************
```

The Alcor pascal system includes a program for patching disk files. This program is used to apply all current and future patches to the pascal system. Initially, patches are supplied to correct for incompatibilities among the various TRS80 operating systems. These patches are contained in files on the release disks.

All patches to pascal should be applied with the patch program since it contains extensive error checking to assure that patches have been entered and applied correctly. To apply patches, follow the following steps:

1. Make a backup copy of the release disk. Do not apply patches to the release disk.
2. The patches should be entered into a file. The initial patches are supplied in this form. Printed patches should be entered into a file using the text editor.
3. Load the disk containing the patch program and execute it by typing "PATCH". If PATCH is a system command (under LDOS for example), then the file PATCH/CMD should be renamed. After the program has been loaded, the disk containing it may be removed.
4. The patch program will prompt for the disk being used for patches. All patches are made using this disk drive. The program will prompt you to change disks when necessary. Enter the disk drive number for the drive that will be used.
5. Enter the name of the file or device for the listing. The patch program will echo the patches to this file or device and will display any error messages there. ":L" will specify the line printer.
6. Enter the name of the file containing the patches. This file must remain on-line during the entire patch process. On two drive systems, this file should be on the system disk. The procedure for single drive systems is covered below. The patches required for both NEWDOS80 and DOSPLUS on the TRS80 model I are contained on release disk number 3 in a file named "NEWDOS/PAT". The patches required for LDOS on the TRS80 model III are on release disk number 2 in a file named "LDOS/PAT".
7. Change disks when prompted to do so. If any errors are detected, error messages will be displayed and the patches will not be applied.

Example Patch Session

The following is an example of how to patch the Pascal System for execution under LDOS on the Model III. A two drive system running TRSDOS 1.3 is assumed. The following steps should be performed before any patching is attempted.

(1) RENAME PATCH/CMD to PATCHP/CMD
    (To prevent conflicts with the system command)
(2) Copy LDOS/PAT to the system disk.
(3) Insert the disk with PATCHP/CMD and invoke PATCHP.

The following information will prompted for at the terminal. Text after the ";" are comments in this manual and will not be present in the terminal session.

```
ENTER DISK DRIVE FOR PATCHES: 1      ; Use drive number 1
LISTING = :L    ;Echo listing to printer  (:D for no device)
ALCOR SYSTEMS DISK PATCH UTILITY 1.0 (C) 1982
PATCHES = LDOS/PAT:0   ;File containing patches on the system disk
LOAD DISK : ALCOR1 INTO DRIVE 1
  PRESS <ENTER> WHEN READY   ;Hit enter to start patching disk1
LOAD DISK : ALCOR2 INTO DRIVE 1
  PRESS <ENTER> WHEN READY   ;Hit enter to start patching disk2

STACK USED = 514 OF 4032 HEAP USED = 1574 OF 29832
TRSDOS READY
```

If :L was used for the listing device then the following listing will appear at the printer.

```
ALCOR SYSTEMS DISK PATCH UTILITY 1.0 (C) 1982
;
;  TRS80 MODEL III FOR LDOS
;
F, RUN/CMD, ALCOR1
P,1A2A,0558,0001,00,13
P,1433,0508,0001,03,00
W,F5A0
;
F, PASCAL/CMD, ALCOR1
P,11CD,0577,0001,00,13
P,0D42,0526,0001,03,00
P,233E,0575,0001,20,4C
W,EFEE
;
F, ED/CMD, ALCOR1
P,11ED,057D,0001,00,13
P,0BD5,055E,0001,03,00
W,F525
;
F, LINKLOAD/CMD, ALCOR2
P,1A2A,0558,0001,00,13
P,1433,0508,0001,03,00
W,F5A0
;
F, PASCALB/CMD, ALCOR2
P,11CD,0577,0001,00,13
P,0D42,0526,0001,03,00
P,246E,0580,0001,20,4C
W,EFE3
E
```

As supplied on diskette, the Alcor Pascal system requires a valid operating system to always be resident in drive 0. It may be one of the following:

MODEL I

* Trsdos 2.3, * Ldos 5.1, Newdos 2.0, Dosplus 3.3, 3.4

MODEL III

* Trsdos 1.3, Ldos 5.1, * Newdos 2.0, * Dosplus 3.3, 3.4

* - These operating systems will function as delivered on diskette without patching. The others require patching with tha Alcor patch program using the control files supplied on diskette. The Pascal system files should be patched under one of the "*" systems and then copied to the desired system format.

Patching on single drive systems

On systems with only one disk drive, all patches will be applied using the system disk. This requires a certain amount of file copying. The patches will be applied a few files at a time. The patch program ("PATCH/CMD") should be copied to a system disk. Several system disks must be created. Each disk will contain the file containing patches (NEWDOS/PAT or LDOS/PAT) and one or more of the pascal files.

Execute the patch program using the procedure above. Once the program has loaded, remove the system disk containing the patch program and insert a system disk containing files to be patched. Specify drive 0 for the patches and proceed with the patching. Error messages will be generated for those files that do not exist on the disk. These messages can be ignored.

SOFTWARE TROUBLE REPORT FORM

Name ____

Address ____

City,State,Zip ____________________________________

Phone ____

Compiler serial number ____________________________

Version number ____________________________________

Operating System __________________________________

* NOTE- A compiler listing of the program at fault must be included. If the problem is associated with the runtime execution then please include any data input and all outputs. Please do not call, but simply send in this STR.

Problem Type

[] Software	[] Editor	[] Linking Loader
[] Documentation	[] Compiler	[] Code generator
[] P-code optimizer	[] RUN	[] Overlayed Compiler
[] Other		

Description of the problem:

Please send to: Alcor Systems
13534 Preston Rd. Suite 365
Dallas, Texas 75240

```
===========================================
!                                          !
!                                          !
!                                          !
!                                          !
!                                          !
!           PASCAL BEGINNERS GUIDE         !
!               Alcor Systems              !
!                                          !
!               Second Edition             !
!               First printing             !
!                   1982                   !
!                                          !
!                                          !
!                                          !
!                                          !
===========================================
```

Copyright (C) 1981 by Alcor Systems

SECOND EDITION
FIRST PRINTING-1982

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Alcor Systems. No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this book, Alcor Systems assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

TRS80 and TRSDOS are trademarks of Tandy Corporation.
USCD Pascal is a trademark of UC Regents, San Diego campus.
Apple is a registered trademark of Apple Computer Inc.
Z-80 is a trademark of Zilog corporation.
CPM is a trademark of Digital Research.

Beginners guide

Introduction

Congratulations on your purchase of the Alcor Pascal System. Alcor Pascal is a powerful language system that will increase the effectiveness of your TRS80 microcomputer. Alcor Pascal is a full implementation of the computer language Pascal with all of its features intact, and therefore compatibility of source programs with other standard Pascal implementations is greatly enhanced. A major effort was expended to produce an efficient and compact Pascal System for your TRS80 computer. Compiled source programs execute between 10 and 50 times faster than many interpreted language systems, and in fact, faster than many Pascals on other systems. Benchmark test results comparing it with other language implementations may be found in the System Implementation Manual. Programs written in Alcor Pascal may be executed on any model TRS80 microcomputer provided the proper runtime package is installed. Runtime support packages are available to licensed Alcor Pascal compiler owners that allow the system to be used on any different model.

Alcor Pascal Basics

The process of developing a program with Alcor Pascal is different than with most interpreted Basic languages. Alcor Pascal is a partially compiled language. This means that the program must be translated into a form that is understandable by the computer before it may be executed. First the source program is entered onto a disk storage file with the aid of the Blaise text editing system, and then translated by the Alcor Pascal compiler into object code modules that may be executed by the host computer. This necessitates the use of disks to store the source program. Although compiled programs may execute on TRS80 computers with less than 48 k bytes of memory and no disk storage, it is recommended that a 48 k system with two mini-floppy disk drives be used for all software development.

Beginners guide

The following items are included in your Alcor Pascal system package:

MANUAL SET

I. BEGINNERS GUIDE
   A quick introduction of how to use this system.
II. PASCAL TUTORIAL
   A step by step introduction to Pascal, aimed at the people with some knowledge of a computer language. All of the Tutorial's example programs are included on diskette in source form. You may compile, modify and execute them while progressing through the Tutorial.
III. Alcor Pascal LANGUAGE REFERENCE Manual
   A detailed guide to the entire Alcor Pascal language.
IV. Alcor Pascal SYSTEM IMPLEMENTATION Manual
   Detailed information on how to use the system.
V. Blaise EDITOR Manual
   A guide on how to use the Blaise text editor.
VI. Quick Reference Programming Card.
VII. Documentation package X-Reference Index.

DISKETTES

I. DISKETTE NO. 1

(a) PASCAL/CMD	Alcor Pascal non-overlayed compiler
(b) RUN/CMD	Run utility for object code execution
(c) ED/CMD	Blaise text editor
(d) ERRORS/DAT	Compiler error message file
(e) HELP/HLP, CMD/HLP, KEY/HLP	editor help files.
(f) T.../PCL	The Tutorial manual example programs in Pascal source form (Model III only)

II. DISKETTE NO. 2

(a) PASCALB/CMD	Alcor Pascal overlayed compiler
(b) PASCAL/OV1-OV4	PASCALB overlay segments
(c) LINKLOAD/CMD	Linking loader and interpreter
(d) TRSLIB/PCL	External declarations for TRS80 library
(e) TRSLIB/OBJ	Object for TRS80 runtime library
(f) STRINGS/PCL	External declarations for STRING library
(g) STRINGS/OBJ	Object for TRS80 string library
* (h) T../PCL	The Tutorial manual example programs in Pascal source form
* (i) DATABASE/PCL	Tutorial Data Base program in source form
* (j) PATCH/CMD	Patch program used to apply updates
* (k) LDOS/PAT	Patch control file to convert system for LDOS execution on the Model III only. Not required for TRSDOS or DOSPLUS.

* Not on Model I version

MODEL I Version only

III. DISKETTE NO. 3

(a) T../PCL	The Tutorial manual example programs in Pascal source form
(b) DATABASE/PCL	Tutorial Data Base program in source form
(c) PATCH/CMD	Patch program used to apply updates
(d) NEWDOS/PAT	Patch control file to convert the system for execution under NEWDOS or DOSPLUS on the Model I. Not required for the Model III.

Beginners guide

Overall system view

The program development process may be visualized by the following diagram:

```
                                 ========
                                !        !
                                !  USER  !
                                 ========
                                    !
                                    !
                                    !       RUN <cmd> OR
                     ED <cmd>       !       LINKLOAD <cmd>
          ==========================!==========================
         !                          !                         !
         !                          !  PAS <cmd>              !
         !                          !                         !
         !                          !                         !
enter    !                translate !                 run     !
Program  !                program   !              program or !
         !                          !              build load !
         !                          !              module     !
         !                          !                         !
  =======V========          ========V========           ======V=====
 !                !        !                 !         !            !
 !  Blaise        !        !   Alcor Pascal  !         !   LOADER   !
 !  TEXT EDITOR   !        !    COMPILER     !         !   PROGRAMS !
  ================          =================           ============
         !                        ^    !                   ^    ^
         !               read     !    !  save             !    !
save     !               source   !    !  object           !    !
source   !               program  !    !  program          !    !
program  !                        !    !                   !    !
         !                        !    !       RUN<cmd>    !    !
         !                        !    !       load object !    !
         !                        !    !       and execute !    !
         !                    =========v=====              !    !
         !                   !               !=============!    !
          ------------------>!    DISKETTE   !                  !
                             !    STORAGE    !<------------------
                              ===============    LINKLOAD <cmd>
                                                 load object and
                                                 build TRSDOS
                                                 /CMD file
```

PLACEMENT OF EDITOR ON DISKETTE

The text editor is supplied on a 5-1/4 " minifloppy diskette. On the TRS80 Model I version, there is not enough free storage for proper operation. The editor and its associated files should be moved to a diskette with at least as much free storage as the size of the file to be edited. The editor system is composed of the main editor file labeled ED/CMD and several help files labeled with the /HLP file extension name. These files may be copied to any diskette for use. The help files contain HELP messages that may be viewed while in the text editor. They are not required to be present. If they are not present on the same diskette as the ED/CMD program, no help information may be obtained during the edit session.

The first action necessary to enter a source program is to invoke the Blaise text editing system. Blaise allows you to enter your program and to modify it as desired. It is a screen oriented text editor that allows you to see exactly what you are editing as you make changes. The edited file may be displayed on the terminal a section at a time and then selectively modified. To use Blaise load diskette no. 1 into the drive and type: ED <file name> The file name should be the name of the file you want to edit in standard TRSDOS notation. If the file name field is left blank, a new file will be created with the name specified at the end of the edit session. If the file name is created with a "/pcl" extension, Alcor Pascal will recognize any file names with this extension as a Pascal source file. This notation will help you keep source programs and translated object programs separated. An example file name is : TEST/PCL:1 If the file name field is left blank the terminal will display:

```
*EOB
```

At this point, you may insert blank lines into the file by depressing the shift key while at the same time depressing the "@" key. Once several blank lines have been inserted into the file, the cursor may be positioned any place on the blank lines by using the arrow keys. To insert the program, simply type it in. The following table lists a few of the editor's key definitions. For a complete explanation of Blaise , see the Blaise reference Manual.

Blaise key definition table

KEY	MEANING
^ !	Cursor up one line
! V	Cursor down one line
<---	Cursor left one character
--->	Cursor right one character
<enter key>	Move cursor to the beginning of the next line.
SHIFT <---	Delete line under cursor.
SHIFT --->	Delete char under the cursor.
CLEAR ^ !	Roll one page towards the beginning of the file
CLEAR ! V	Roll one page towards the end of file
SHIFT @	Insert a blank line before the line under the cursor.
CLEAR C	Enter command mode

* Note - See BLAISE Reference Manual for complete information on key definitions and commands.

Beginners guide

Entering a program

Blaise

The thing to remember at this point is, what is displayed will be stored on the disk file. By moving the cursor to any position, and typing characters, text may be inserted into the file. If a character is overstruck, the old character will be replaced by the new one. This overwrite mode is the default mode in the editor. The following program may be typed in at this point.

```
PROGRAM test;
BEGIN
WRITELN('* I AM A PASCAL WIZARD');
END.
```

Once this program has been correctly entered, depress the clear key followed by the character C . This will cause Blaise to enter the command mode. A pair of angle brackets should appear at the bottom left side of the screen. Type the word exit. Then depress the enter key. If any errors are made in typing exit, they may be corrected by backspacing over the error and retyping. A backspace may be performed by depressing left arrow key. After the enter key is depressed, a prompt for the file name will appear. You may specify any legal TRSDOS file name including the drive specifiers. Depress the enter key and the file will be saved. The source program may now be translated by Alcor Pascal .

Compiling the program

Once the program has been entered into the computer and placed in a disk file, the next step is to compile it. The pascal compiler translates the source program into a form that the computer can execute. For example, suppose that you have entered the previous example. This program may be stored in a file called: TEST/PCL The simplest method to execute this program is to type the two commands:

```
PASCAL TEST
RUN TEST
```

This will compile and execute your program. Let's examine the process in more detail. The first line causes the operating system to load and execute the pascal compiler. The compiler then translates the pascal source code contained in the file: TEST/PCL into code that can be run on the computer. This code is stored in a file called: TEST/OBJ . A listing will be sent to the terminal. The listing shows the source program and will contain error messages for any errors detected. The listing will be described in more detail in the System Implementation Manual. If errors are detected, code numbers and error messages will be contained in the listing. The errors in the source program must be corrected before the program can be executed.

Running the program

Once the program has been compiled without errors, it can be executed with the "RUN" command. "RUN TEST" causes the object code stored in the file: "TEST/OBJ" to be loaded into memory and executed The first thing that a pascal program normally does is to open the files "INPUT" and "OUTPUT". When this happens, the prompt:

```
INPUT  =

OUTPUT =
```

will appear on the screen. At this time you may enter the file or device to be used when the program writes to input or output. If you simply press the enter key, then input/output will be directed to the terminal. When any file is opened by a pascal program (by calls to RESET or REWRITE), a prompt will appear on the screen. To the left of the equal sign will be the name of the file being opened. You should type the name of the disk file or device to be associated with that file.

Running the program

The runtime mapping of pascal files to physical files and devices allows a program to redirect its input and output without any changes to the source program and without recompiling the program. For example, you could execute the TEST program with the output going to the screen. When you are satisfied with the results, the output can be directed to a file or line printer instead.

The file names that you type to direct pascal input and output are in the same format as normal TRSDOS file names. The disk drive specification is optional as in TRSDOS. There is one extension : Input and output to any pascal file can be sent to physical devices as well as to a file. The device names are simple extensions to the disk names used by TRSDOS. For example, the name of the line printer is ´:L´, and the name of the crt is ´:C´. There is also a dummy device. If a file is associated with ´:D´, then no actual output occurs. This is useful if you wish to run the program and discard some of its output. Complete information concerning how to compile, link and run Pascal programs may be found in the Alcor Pascal System Implementation Manual.

BLAISE TEXT EDITOR
Alcor Systems

Second edition
First printing
1982

Copyright (C) 1981 by Alcor Systems

SECOND EDITION
FIRST PRINTING-1982

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Alcor Systems. No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this book, Alcor Systems assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

TRS80 and TRSDOS are trademarks of Tandy Corporation.
USCD Pascal is a trademark of UC Regents, San Diego campus.
Apple is a registered trademark of Apple Computer Inc.
Z-80 is a trademark of Zilog corporation.
CPM is a trademark of Digital Research.

INTRODUCTION

A text editor is simply a program that is used to input textual data into a computer system in such a way that the information may be saved and used at a later time. Usually, text editors may be divided into two categories. They are either line or screen oriented. Most TRS80 users are familiar with line oriented type because this is the kind of editor that is supplied with the TRS80 basic system.

A line oriented text editor is characterized by its many commands required to simply delete or insert a character. Also, as movement through a document or program is performed, the screen is not updated to reflect the current position in the storage file. If it is desired to view the current line or page being edited, you must type a command to view it. This hidden context of the file being edited makes it hard to visualize changes, and remember what text is being changed.

A screen oriented text editor is characterized by its typewriter like display of text. The screen usually is kept updated with the current page of text that is being edited. To perform simple character insertions or deletions, you simply move the blinking cursor on the screen to the desired character and hit the delete character or insert character key. Many other editing commands that are difficult to use in a line text editor become very simple in a screen oriented text editor. For example, scrolling the display through the file may be performed by the touch of a key. Most software word processors are simply powerful text editors. Word processors usually have additional commands to insert information into the file such that when the file is printed on a line printer, the page will have the desired format. Needless to say, a good screen oriented text editor will make program entry or document preparation much easier and faster.

The Alcor Systems Blaise Text Editor provides a convenient tool for entering programs or other textual documents into any computer system. Blaise is compatible with most TRSDOS files and may be used for entering or editing: Basic source programs, Pascal source programs, letters or any other written documents.

(C) copyright 1981 Alcor Systems

GETTING STARTED

The Alcor Systems Text Editor requires:

HARDWARE

(1) Model I or III TRS80 computer
(2) 48K of memory
(3) 1 disk drive

SOFTWARE

(1) TRSDOS, or TRSDOS compatible operating system. (Not supplied)

PLACEMENT OF EDITOR ON DISKETTE

The text editor is supplied on a 5-1/4 " minifloppy diskette. The editor system is composed of the main editor file labeled ED/CMD and several help files labeled with the /HLP file extension name. These files may be copied to any diskette for use. The help files contain HELP messages that may be viewed while in the text editor. They are not required to be present. If they are not present on the same diskette as the ED/CMD program, no help information may be obtained during the edit session.

EDITOR WORK FILE

The text editor creates a new text file that is a copy of the file being edited. All editing changes are made to this work file. (T011/TMP - TRS80) This is a protection feature to prevent any fatal hardware or software errors during the edit from destroying the edited file. After the editor successfully exits, and consistency checks have been made, the work file becomes the new copy of the edited file. The old copy is then deleted. This requires that there always be a minimal amount of disk space available on the diskette where the work file exists. If during an exit the drive on which the work file, or the original file is placed runs out of disk space, the editor will flag an error message allowing an abort or appropriate actions to be taken.

TYPE OF FILES THAT MAY BE EDITED

New or old files may be created or edited. The size of files that may be edited is only limited by the disk space available on a diskette. Files may not be split across diskette boundaries, i.e.; the whole file must reside on a single diskette. The file name syntax, i.e.; legal naming conventions are the same as that allowed by the particular operating system in use. The text editor files are compatible with: Normal TRSDOS Basic files, Alcor Pascal source and object files, or any other ASCII formatted files that comply with TRSDOS file conventions.

REMOVAL OF DISKETTES FROM DRIVES DURING AN EDIT SESSION

The diskette containing the ED/CMD file may be removed after an edit session has begun, subject to the following restrictions :

(1) There always must be a diskette with an operating system installed in the designated system drive. This diskette may be swapped during the edit session as long as the new diskette contains a valid operating system, and the change does not violate (2)-(4).

(2) Removal does not cause the file containing the editor workfile to be removed.

(3) If help files are removed, then no HELP messages will be available.

(4) Before exiting the editor, or appending lines to the text buffer, the diskette containing the origional file is replaced in the drive.

* If the above rules are followed, you may change diskettes in order to use the INSERT FILE command in the editor.

FUNDAMENTALS EDITOR COMMANDS

Editor commands may be accessed by two different methods. One is by hitting certain pre-defined key sequences, and the other is by entering a mode that allows command names to be typed in.

For most commands there are two alternate key sequences that will invoke the command. One is a control key sequence and the other is a labeled key sequence.

A control key sequence on the TRS80 MODEL III is initiated by simultaneously holding the shift and down arrow keys depressed and then hitting one of the printable characters "A-Z" . Control key sequences have the advantage that they are in effect single key stroke operations. I.E.; if the Shift and down arrow keys are held down, you may quickly strike any other key repeatedly such as a tab key. This would be preferable to hitting the shift key and then the tab key. That would be two keystrokes per movement. The TRS80 keyboard does not lend itself to single keystroke commands.

A labeled key sequence is initiated by : (1) simply hitting a labeled key. (2) Hitting the clear key then a labeled key. (3) Holding the shift key down and then hitting a labeled key.

(C) copyright 1981 Alcor Systems

ANY command that may be invoked by a keystroke sequence may be invoked by entering it's command name. There are certain additional commands that are only accessible in the the command name mode. They primarily are used for setting editor parameters or for performing commands that require prompted input from the user. For all following discussions, the mode which allows key sequence commands shall be referred to as the compose mode, and command name entry as the command mode.

All editing is performed on text that is loaded into a RAM text buffer. This buffer holds new text data entered or may append text data from an existing TRSDOS file. As editing is performed, the updated buffer may be written to the file as desired.

HOW TO START

To invoke the text editor simply type the TRSDOS command:
ED filename

NEW FILES

To create a new file, leave the filename field blank. The editor will clear a new text buffer. The screen will be cleared and will display:

*EOB

at the top left hand side. A cursor will be displayed in column one of the display. The editor automatically enters the compose mode upon startup. A blank line must be inserted into the empty text buffer before any text may be inserted. A blank line may be inserted by depressing the shift key while at the same time hitting the key labeled "@" . The blank line will always be inserted before the cursor line. After several blank lines have been inserted into the buffer, text may be typed directly into the buffer. The text is always entered into the buffer beginning at the position of the cursor. If the cursor is positioned over any text, as each new character is entered, the old character is overwritten. If text is to be inserted in the middle of an existing line, simply depress the clear key followed by the I key. Any new text typed will be inserted at the cursor position causing the rest of the line to move to the right. The insert setting is canceled when any command such as a cursor movement is performed.

COMPOSE MODE

CURSOR MOVEMENT

The following denotes a key sequence.
Shift/key ====> depress shift and key simualtaneously.
Clear/key ====> Hit clear key then labeled key.

The cursor may be positioned any place on the screen by depressing any of the following labeled keys while in the compose mode.

^	Cursor up one line. If the cursor is at the top of screen, then the screen scrolls one line.
v	Cursor down one line. If the cursor is at the bottom of the screen, then the screen scrolls one line.
--->	Cursor to the right one character. If cursor is at the right edge of the screen, then no movement occurs.
<---	Cursor to the left one character. If the cursor is at the left edge of the screen, then no movement occurs.
enter	Cursor to the beginning of the next line.
Clear/ <---	Cursor to the beginning of line.
Clear/ --->	Cursor to the end of line.
Clear/ ^	Scroll the display one page toward the beginning of file.
Clear/ v	Scroll the display one page toward the end of the file.
Clear/T	Tab right. If the cursor is at the right edge of the screen, then the cursor wraps around to the leftmost tab stop.
Clear/B	Tab left. If the cursor is at the left edge of the screen, then the cursor wraps around to rightmost tab stop.
Clear/H	Positon cursor to the top left of display. (Home)

TEXT DELETION

Shift/ --->	Delete character under the cursor.
Shift/ <---	Delete the entire line under the cursor.
Clear/K	Delete text from cursor to the end of the line.

TEXT INSERTION

Clear/I	Enter insert character setting. Will overide the text overwrite setting. This allows text to be inserted anyplace in an existing text line. Place the cursor at the desired location and initiate the the Clear/I sequence and type in the text.
Shift/@	Insert a blank line into the text buffer.
Clear/D	Duplicate the line above onto the cursor line. Text to the right of the cursor position is replaced by a copy of the text on the line above.

EDITOR PARAMETERS AND SETTINGS

Clear/A	Toggle the auto indent setting. Auto indent causes the enter key to align the cursor with the first non-blank on the next line. If the next line is blank, the cursor is placed below the first non-blank on the line above. This feature is useful when Pascal programming with indentation.
Clear/S	Sets the typewriter like tab stop at the current cursor position.
Clear/Y	If a tab exists at the current cursor position then it is cleared.
Clear/?	Display the amount of unused memory available for text buffering.
Clear/C	Enter command name mode.

(C) copyright 1981 Alcor Systems

TEXT MODIFICATION

Clear/G	Merge the line after the cursor with the cursor line.
Clear/O	Split the cursor line into two lines at the cursor position.
Clear/F	Search forward in the text buffer for the next occurrence of the string in the find string buffer. The find string buffer is loaded with the search string in the command mode by the FIND command.
Clear/R	Search forward in the text buffer for the next occurrence of the string in the find buffer and replace with the contents of the replace buffer. The buffers are loaded with strings in the command mode by the REPLACE command.

COMMAND MODE

If while in the compose mode a Clear/C key sequence is initiated, Blaise will enter the command mode. A pair of angle brackets along with the cursor should appear in the bottom left edge of the display. Any command that is defined by a key sequence may now be invoked by typing its two character mnemonic command name. Mnemonic command names for all of the key sequence commands are listed in the command table summary. There are 16 additional commands that are available in command mode that are not accessable in the compose mode. They are primarily used for setting editor parameters or require prompted information from the user. After a command has been executed in the command mode, Blaise will automatically re-enter the compose mode, and place the cursor at it´s new position in the text. Command name abbreviations are allowed for all commands.

When in command mode, the command line may be edited using the left arrow key. Pressing the left arrow will delete the last character on the line and move the cursor back one column. Command entry may be aborted by pressing the shift/left arrow. This will return the editor to compose mode.

TEXT BUFFER MANAGEMENT
PRE-EXISTING FILES

If the file to be edited already exists when Blaise is invoked, it's first action will be to load the text buffer with a portion of the file. All editing of a file is performed to text that is stored in the text buffer. Blaise has enough text buffer area to load approximately 13 K bytes of text data at a time on a 48 K machine. Upon start-up, Blaise will not load the entire buffer with data, but preserves a preset amount for new lines and changes. If the section of the file to be edited is not loaded into the text buffer, as evidenced by scrolling through the buffer, it must be appended to the end of the buffer with the APPEND command. During the scroll operation, when the end of the current buffer is encountered, an "*EOB" message will be displayed. Any desired number of lines may be appended to the end of the buffer. Appending lines to the buffer does not erase any current buffer data, but simply adds to it, therefor it is possible for memory to become exhausted. If the buffer memory is exhausted as evidenced by a memory message during an APPEND operation, a portion of the text buffer must be written to the workfile to release some buffer space. This is accomplished with the WRITE command. The write command will write the specified number of lines from the beginning of the text buffer to the workfile. Once these lines are written, they may not be loaded into the buffer again during the current edit session. By writing and appending to the text buffer, any size file may be edited by Blaise. In fact, the file size is limited only by the size of a file that may fit on a diskette, if the following precautions are taken.

WORK FILE

As previously discussed, all editing is performed on the editor work file. If the work file is placed on the same disk as the original file, then there must be enough space on the diskette at all times for two copies of the edited file. This effectively cuts the size of file that may be edited by one-half. The placement of the workfile on a specific drive and diskette is dependent on the how Blaise is invoked. Under TRSDOS, if the file name drive specifier is not appended to the file name upon bidding of the editor, Blaise will search the various drives, and place the work file on the lowest numbered drive encountered that has enough space. If the drive specifier is appended to the file name upon bidding of the editor, then the work file will be placed on the same diskette as the original file.

Under TRSDOS, if the file to edited is a new file, I.E.; the file name field is left blank, the workfile will be placed on the lowest numbered drive that has enough space.

COMMAND PARAMETERS

Some commands that may be accessed in command mode require user input parameters for execution. All such commands may be invoked by two different methods in the command mode. Beginners may simply type the command name and then hit the enter key. If any parameters are required, Blaise will prompt the user for the parameters. All parameter entries should be terminated by the user by hitting the enter key. Advanced users may desire to enter the parameters after the command name on the same line. If all of the parameters are entered on the same line as the command name, hitting the enter key will cause Blaise to immediately execute the command. If all of the parameters were not entered on the command line, Blaise will prompt te user for the remaining unspecified inputs. The rule for entering command parameters on the command line is that any string parameter such as a file name must be enclosed or delimited by double quotes. An example would be "file name" .

COMMAND EXPLANATIONS

The sixteen additional commands available in the command mode are explained below. Where parameters are required, the command line form is included.

APPEND numberoflines

The append command reads text from the original file and appends it to the end of the text buffer. After the append executes, a message will appear at the bottom of the display with the total number of bytes available for additional text. If the memory becomes exhausted, then text must be written to the work file by the WRITE command.

WRITE numberoflines

A WRITE command will write to the work file the specified number of lines starting from the first line of the text buffer. As the write occurs, buffer space will be released. Once the specific lines have been written to the workfile, they may no longer be edited during the current session. They are permanently saved in the workfile.

HELP topic

The help command will display help messages to the screen concerning the specified topic if help information is available. Supplied help topics include: HELP - General help information. CMD - Command mode information. KEY - Key definitions and compose mode information. If the subject is left blank, general help will be displayed. The help information may be viewed by using the same movement commands used in the showfile command. To exit, hit clear followed by "C".

(C) copyright 1981 Alcor Systems

SHOWFILE "filename"

The showfile command will open the desired file and display a portion of it. Several special commands may be issued while in the showfile command. They are:

```
============================================================
!                                                          !
!  Clear/   ^                 Scroll the display up        !
!           !                 one page in the file.        !
!                                                          !
!  Clear/   !                 Scroll the display down      !
!           v                 one page in the file.        !
!                                                          !
!       linenumber           Position absolutely to        !
!                            line number in file.          !
!                                                          !
!      + linenumber          Scroll specified number       !
!      - linenumber          of lines relative to current  !
!                            cursor line. + rolls to the   !
!                            end of buffer, - rolls to the !
!                            beginning of buffer.          !
!  Clear/C                   Return to editing             !
============================================================
```

SHOWLINE linenumber

The showline command is convenient for positioning absolutely to any line number in the buffer. Showline 1 is a quick way for positioning to the beginning of buffer, and Showline 9999 for positioning to the end of buffer.

INSFILE "filename" startline numberoflines

This command will insert a portion of any pre-existing file into the text buffer starting at the cursor location.

FIND "string"

The find command will search forward in the text buffer, (starting at the cursor position) for the specified string. If the string is found, cursor will be positioned at the first occurrence, at the beginning of the string. If the string is delimited by quotes, then leading or trailing blanks will be included in the search string.

REPLACE "old string" "new string"

Replace will search forward in the text buffer for the old string. (starting at the cursor position) If the old string is found, then the first occurrence will be replaced with the newstring, and the cursor repositioned at the beginning of the new string.

QUOTE "string"

The quote command is used when it is desired to insert some non-printable character into a file. It is also useful for inserting certain printable characters that are not on the TRS80 keyboard into the file. The quoted string is inserted at the current cursor position. Non-printable characters may be represented by a # followed by the two character hexadecimal number. (ASCII representation) For example, #5B is the left bracket.

\+ numberoflines
\- numberoflines

The plus and minus commands are used to position the cursor a specified number of lines relative to the cursor line. (The blank after the + or - is required)

ROLL numberoflines

Roll will set the page size for all scrolling commands. It is set to 13 lines by default upon editor invocation.

HSCROLL column

The hscroll command will scroll the display horizontally to the left or right. This feature allows editing of files wider than the TRS80 screen. Once a horizontal scroll is performed, the display will remain in this mode until repositioned by another horizontal scroll command. The column parameter is the new column position for the left edge of the screen. Max column for TRS80s is 16.

TABS integer
TABS = integer,integer,integer.....

TABS integer will set a tab stop every integer positions. All tabs may be cleared by setting integer to 0. The editor defaults to a tab stop every third character position. An alternate form of the command exists. Tabs may be defined as: TAB = integer,integer,integer.... at the specified columns.

QUIT "answer yes/no"

Will abort an edit session. All changes to the original file are lost, and the original file is preserved.

EXIT "filename"

The exit command is used to successfully end an edit session. The exit command will write out the entire text buffer to the workfile. Any non-appended lines from the original file will also be written to the workfile. Once consistency checks have been made, file renaming or deletions will occur. If a filename was specified when entering the editor, you can respond with the enter key to the filename prompt. This will replace the old file that was edited. You can also specify the name of a new file. If the filename was left out when bidding the editor, a filename must be specified in the exit command.

EDITOR CONTROL KEYS

KEY	CONTROL	MNEMONIC	FUNCTION
^	CTL/U	UP	Cursor up
v	CTL/J	DN	Cursor down
<	CTL/H	LF	Cursor left
>	CTL/R	RT	Cursor right
CLEAR ^	CTL/B	RB	Roll backward
CLEAR v	CTL/A	RF	Roll forward
CLEAR <		BL	Cursor to BOLN
CLEAR >		EL	Cursor to EOLN
SHFT >	CTL/P	DC	Delete character
SHFT <	CTL/N	DL	Delete line
SHFT @	CTL/O	IL	Insert line
ENTER	CTL/M	NL	New line
CLEAR A		AI	Toggle auto indent
CLEAR B	CTL/T	BT	Back tab
CLEAR C		CM	Command mode
CLEAR D	CTL/D	DU	Duplicate line
CLEAR F	CTL/C	FN	Find next string
CLEAR G		MG	Merge two lines
CLEAR H	CTL/L	HM	Home
CLEAR I	CTL/Q	IC	Insert char mode
CLEAR K	CTL/K	DE	Delete to EOLN
CLEAR O		OL	Open line at cursor
CLEAR R	CTL/Z	RN	Replace next string
CLEAR S	CTL/W	ST	Set tab
CLEAR T		TB	Tab
CLEAR Y		CT	Clear tab
CLEAR ?		MM	Display memory

EDITOR COMMANDS

APPEND	Add text to buffer
EXIT	Exit and save file
FIND	Find string
HELP	Display help
HSCROLL	Horizontal scroll
INSFILE	Insert file
QUIT	Abort changes
QUOTE	Insert literal string
REPLACE	Replace string
ROLL	Set roll
SHOWFILE	Display a file
SHOWLINE	Position to line
TABS	Set tabs
WRITE	Write text to file
+	Move forward by lines
-	Move backward by lines

(C) copyright 1981 Alcor Systems

DEVICE NAMES

Name	Device
:L	Line printer
:C	Screen
:D	DUMMY

ASCII Character Set

Dec-imal	Hex	Name	Dec-imal	Hex	Name
0	00	NUL	11	0B	VT
1	01	SOH	12	0C	FF
2	02	STX	13	0D	CR
3	03	ETX	14	0E	SO
4	04	EOT	15	0F	SI
5	05	ENQ	16	10	DLE
6	06	ACK	17	11	DC1
7	07	BEL	18	12	DC2
8	08	BS	19	13	DC3
9	09	HT	20	14	DV4
10	0A	LF	21	15	NAK

ASCII Character Set

Decimal	Hex	Name	Decimal	Hex	Name
22	16	SYN	75	4B	K
23	17	ETB	76	4C	L
24	18	CAN	77	4D	M
25	19	EM	78	4E	N
26	1A	SUB	79	4F	O
27	1B	ESC	80	50	P
28	1C	FS	81	51	Q
29	1D	GS	82	52	R
30	1E	RS	83	53	S
31	1F	US	84	54	T
32	20	" "	85	55	U
33	21	!	86	56	V
34	22	"	87	57	W
35	23	#	88	58	X
36	24	$	89	59	Y
37	25	%	90	5A	Z
38	26		91	5B	[
39	27	'	92	5C	\
40	28	(	93	5D	]
41	29	)	94	5E	^
42	2A	*	95	5F	
43	2B	+	96	60	`
44	2C	,	97	61	a
45	2D	-	98	62	b
46	2E	.	99	63	c
47	2F	/	100	64	d
48	30	0	101	65	e
49	31	1	102	66	f
50	32	2	103	67	g
51	33	3	104	68	h
52	34	4	105	69	i
53	35	5	106	6A	j
54	36	6	107	6B	k
55	37	7	108	6C	l
56	38	8	109	6D	m
57	39	9	110	6E	n
58	3A	:	111	6F	o
59	3B	;	112	70	p
60	3C	<	113	71	q
61	3D	=	114	72	r
62	3E	>	115	73	s
63	3F	?	116	74	t
64	40	@	117	75	u
65	41	A	118	76	v
66	42	B	119	77	w
67	43	C	120	78	x
68	44	D	121	79	y
69	45	E	122	7A	z
70	46	F	123	7B	{
71	47	G	124	7C	|
72	48	H	125	7D	}
73	49	I	126	7E	~
74	4A	J	127	7F	DEL

(C) copyright 1981 Alcor Systems

System Implementation Manual
TRS -80- MODELS I , III
Alcor SYSTEMS

Second Edition
First Printing
1981

Copyright (C) 1981 by Alcor Systems

SECOND EDITION
FIRST PRINTING-1982

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Alcor Systems. No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this book, Alcor Systems assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

TRS80 and TRSDOS are trademarks of Tandy Corporation.
USCD Pascal is a trademark of UC Regents, San Diego campus.
Apple is a registered trademark of Apple Computer Inc.
Z-80 is a trademark of Zilog corporation.
CPM is a trademark of Digital Research.

INTRODUCTION

This manual describes the specific characteristics of Alcor Pascal as implemented on the TRS -80- models I and III microcomputer. The implementation of Alcor Pascal on the two different models is the same except where noted. In every language system implementation, there are certain language features that vary upon computer implementation. One of the advantages of Pascal is that these variations are minor, and if a programmer minimizes the use on non-standard language extensions, source programs may be written that have a high degree of portability. Other machine dependent characteristics are such items as how to invoke the compiler and support utilities.

The process of building an executable pascal program may be summarized by the following diagram.

(C) copyright 1981 Alcor Systems

System overview diagram

```
                ================
               !                !
               !   Source file  !
                ================
                       V
             ====================
            !                    !
            !  Compiler program  !
             ====================
                       V
             =====================
            !                     !
            !  Translated object  !
            !        file         !
             =====================
                       V
 .......................................
:                                       :
:     Optional development package      :
:                                       :
:     (1) Optimizer                     :
:     (2) native code generator         :
:.......................................:
                       V
             ===================
            !                   !
            !   Loader program  !
             ===================
                       V
         ===========================
        !                           !
        !   User program executing  !
        !   in computer memory      !
         ===========================
                     ^  !
                     !  v
         ===========================
        !                           !
        !   Input/Output files for  !
        !       user program        !
         ===========================
```

System Description

Pascal Compiler

The Alcor Pascal compiler is simply a program that is written in Pascal and that executes on the host computer. It's purpose is to translate other Pascal source programs into an intermediate language called P-CODE . The p-code is a low level language designed specifically as a target language for the pascal compiler and resembles the assembly language for a stack oriented computer. Once a program has been compiled, the object p-code program is stored in an intermediate file. The intermediate file may be loaded and executed by the host computer or run through the optional development package.

Optional development package
(not supplied with Alcor Pascal)
Optimizer program

After the source program has been translated into object code, it may be processed by the OPTIMIZER. The purpose of the optimizer is to remove statement redundancy in the translated object code. This will effectively reduce the final size of the program by approximately 25-30 per-cent. The optimizer should be used where program size is important. The optimized p-code is an exceptionally compact representation of the pascal program. This is evidenced by the fact that the Pascal compiler itself (an 8500 line pascal program), can be run on a 48k machine without resorting to overlays.

Codegen program

If program execution speed is important, the native code generator (codegen) program may be used to process the object program file. Codegen will generate native Z-80 code which may be directly executed by the processor. Execution speed is usually increased by a factor of 3 - 5 times. One of the drawbacks of code generation is that the resultant program will grow in object code size by a factor of 2 - 3 over the p-code version. For large Pascal programs, (such as the compiler itself) the resultant program image may not fit into available memory. For small programs this may not be a factor. To combine the best of both worlds the optional codegen program will allow selective code generation of specific modules in a program. This allows the critical paths of a program to be translated into native Z-80 instructions, while at the same time reducing the overall program size by utilizing p-code for the rest of the program. If program size is not a factor, full code-generation may be performed.

(C) copyright 1981 Alcor Systems

System Description

Standard linking loader

Included in the standard release package is the linking loader. After the compiler has translated the source code into p-code, the p-code file may be loaded into memory and executed. The program that performs this is the LINKING LOADER. Its purpose is to load any number of object modules into memory. This allows separate compilation of procedures and functions. To perform separate compilation of a program, procedure or function, the compiler NULLBODY option must be used. For more information, see the Alcor Pascal Reference Manual. The linking loader includes an interpreter in the final load module that executes the p-code instructions when the program is run. The linking loader also has the capability of storing the memory image of the program as an executable command file. Once an image has been saved, the program can be executed simply by typing the file name at the TRSDOS command level. The linking loader and interpreter is a 9000 line program written in Z80 assembly language.

ALCOR PASCAL EFFICIENCY

A benchmark program composed of Pascal source statements was used to measure the efficiency of programs translated by Alcor Pascal. The benchmark program used was published by BYTE MAGAZINE, SEPTEMBER, 1981 . It is an unbiased benchmark program and represents a cross section of what any Pascal implementation should execute efficiently. The results clearly indicate Alcor Pascal's superiority to many other Pascal implementations. The following tables represent a summary of the author's findings along with the data for Alcor Pascal added.

(C) copyright 1981 Alcor Systems

Results of Benchmark published in BYTE 09/81, page 182

(With Alcor Pascal added)

Interpreted languages

Language and machine	Compiled bytes	Total bytes	Compile load (secs)	Execute (secs)
Alcor Pascal (4mhz Z-80)	300	12,754	21	173
UCSD Pascal (4 mhz Z-80)	282	8282	====	239
UCSD Pascal TRS-80-model II (4mhz Z-80)	282	8282	60	274
Alcor Pascal TRS-80-model III (2 mhz Z-80)	300	10,568	44	344
Alcor Pascal TRS-80-model I (1.7 mhz Z-80)	300	10,549	60	403
Pascal/M, (4 mhz Z-80)	301	21,933	50	450
JRT Pascal, (4 mhz Z-80)	232	11,498	65	470
UCSD Pascal, Apple II (6502)	287	=====	43	516
Microsoft MBASIC, (4 mhz Z-80)	====	=====	====	2250
Apple integer basic, 6502	====	=====	====	2320
Applesoft (real) , 6502	====	=====	====	2806
Level II BASIC using integers TRS-80- model I (1.7 mhz)	371	6625	9	4753
Microsoft Cobol version 2.2, Z-80	786	17,605	146	5115
Level II BASIC using real TRS-80- model I (1.7 mhz)	371	6625	9	5404

* Microsoft MBASIC is a trademark of Microsoft

(From BYTE September, 1981)

Native code Pascal

Language and machine	Compiled bytes	Total bytes	Compile load (secs)	Execute (secs)
* Pascal MT + (4 mhz Z-80)	308	3043	102	19
Alcor Pascal (4mhz Z-80)	851	13,305	48	49
Alcor Pascal TRS-80-model III (2.0 mhz Z-80)	851	11,119	96	98
* Ithaca Intersystems Pascal/Z (4 mhz Z-80)	761	3328	124	109
Alcor Pascal TRS-80-model I (1.7 mhz Z-80)	851	11,100	114	114

* NOTE- LOAD MODULES DO NOT INCLUDE FLOATING POINT AND MOST I/O RUNTIME.

* Pascal MT is a trademark of Micro Systems
* Pascal Z is a trademark of Ithaca Intersystems

Overlayed pascal compiler

The size of a Pascal program that may be compiled is dependent on the number of symbols used in the source program and not necessarily the number of lines in the program. The non-overlayed compiler should be able to compile a typical 1000 line program with all of its associated symbols. A further improvement can sometimes be made by separately compiling procedures or functions. If the program is too large for the non-overlayed compiler, the overlayed compiler may be used. The overlayed compiler has been segmented such that parts of it reside on the disk during execution, and are read into memory only as needed. The overlayed compiler will execute more slowly than the non-overlayed version, but generates identical object code. The overlayed compiler has enough space to compile a typical 4000 line Pascal program with all of its associated symbols.

Using Alcor Pascal on the TRS80

The first step in developing a computer program is to define the problem and develop algorithms for the solution. Once the algorithms are specified, the next task is to translate them into a programming language. Pascal is a particularly good expression language for a large number of problems. The program is then entered into the , computer and executed. This section describes the procedures for performing the last two steps on the TRS80. If you are not familiar with the Pascal language, refer to the Alcor systems pascal tutorial for information on the language. For those familiar with pascal, the pascal reference manual contains compact detailed information on the features of Alcor pascal.

Once the program has been designed, the next step is to enter the program into the computer. This is normally accomplished with the aid of a text editor. A screen oriented text editor is supplied with the compiler. For details on how to use this editor refer to the Editor reference manual.

Compiling the program

Once the program has been entered into the computer and placed in a disk file, the next step is to compile it. The pascal compiler translates the source program into a form that the computer can execute. For example, suppose that you have developed a program to prepare your income tax return. This program may be stored in a file called: TAXES/PCL. The simplest method to execute this program is to type the two commands:

```
PASCAL TAXES
RUN TAXES
```

This will compile and execute your program. Let´s examine the process in more detail. The first line causes the operating system to load and execute the pascal compiler. The compiler then translates the pascal source code contained in the file: TAXES/PCL into code that can be run on the computer. This code is stored in a file called: TAXES/OBJ . A listing will be sent to the CRT. The listing shows the source program and will contain error messages for any errors detected. The listing will be described in more detail in a later section. If errors are detected, code numbers and error messages will be contained in the listing. The errors in the source program must be corrected before the program can be executed.

Once the program has been compiled without errors, it can be executed with the "RUN" command. "RUN TAXES" causes the object code stored in the file: "TAXES/OBJ" to be loaded into memory and executed.

Using Alcor Pascal on the TRS80

The first thing that a pascal program normally does is to open the files "INPUT" and "OUTPUT". When this happens, the prompts:

```
INPUT  =
OUTPUT =
```

will appear on the screen. At this time you may enter the file or device to be used when the program writes to input or output. If you simply press the enter key, then input and output will be directed to the screen. When any file is opened by a pascal program (by calls to RESET or REWRITE), a prompt will appear on the screen. To the left of the equal sign will be the name of the file being opened. You should type the name of the disk file or device to be associated with that file.

The runtime mapping of pascal files to physical files and devices allows a program to redirect its input and output without any changes to the source program and without recompiling the program. For example, you could test the taxes program with the output going to the screen. When you are satisfied with the results, the output can be directed to a file or line printer instead.

The file names that you type to direct pascal input and output are in the same format as normal TRSDOS file names. The disk drive specification is optional as in TRSDOS. There is one extension. Input and output to any pascal file can be sent to physical devices as well as to a file. The device names are simple extensions to the disk names used by TRSDOS. For example, the name of the line printer is ':L', and the name of the crt is ':C'. There is also a dummy device. If a file is associated with ':D', then no actual output occurs. This is useful if you wish to run the program and discard some of its output.

THE PASCAL COMMAND

The PASCAL command causes the pascal compiler to be loaded and executed. This command has several forms. The simplest form is:

(angle brackets required when stack is specified)

```
PASCAL <stack>  file name
```

where file name is the name of a file containing a pascal program. The <stack> is an optional parameter that sets an upper limit on memory space that the compiler may use for stack manipulations. The default size should be suitable for most applications. The default stack size is 3.5K for the non overlayed compiler. 4.5K should be suitable for most large programs. If more stack is allocated than available, the compiler may terminate prematurely with unpredictable results. The compiler itself is a pascal program and follows the same conventions for stack and heap usage as other Pascal programs. (See pages 9,10) In the short form, the extension for the source file is assumed to be /PCL and the object code is sent to file name/OBJ. Any extension typed in the command line will be ignored. A disk drive name may be specified. For example,

```
PASCAL TAXES:1
```

will cause the program "TAXES/PCL:1" to be compiled and the object to be stored on disk drive one. In this case the same disk drive will be used for both source and object. If the disk drive is omitted, TRSDOS will search all the drives for the first occurrence of TAXES/PCL and will store the object on the lowest numbered drive with space available. In the short form, the listing will always be displayed on the crt screen.

The long form of the pascal command uses simply: PASCAL to invoke the compiler. In this case, the file names for the source, object and listing will be prompted for on the screen. You should type the name of the actual files to be used. Normal TRSDOS syntax applies. In this case the file names are used as specified. The source and object can be on different disk drives and the listing can be placed in a file, sent to the screen or sent to the line printer. For example, the following sequence will cause the file: "TAXES/TMP" to be compiled with the object code stored in "TAXES/OBJ" on disk drive 2 and the listing will be sent to the line printer.

```
PASCAL  <stack>
SOURCE  = TAXES/TMP
OBJECT  = TAXES/OBJ:2
LISTING = :L
```

THE RUN COMMAND

The run command is used to load and execute a previously compiled pascal program. The object code will be loaded and the program executed. The run command contains the object code for the TRS80 support routines (such as SETPOINT, CLEARSCREEN, etc). Any of these routines can be called. If any other external procedures are required, the linking loader must be used to link these external procedures to the program. The run command is invoked as:

```
RUN program
```

Pascal programs use a stack to store local variables and to save return addresses for procedure and function calls. This stack is allocated when the program is run and the required size is determined by the number and type of variables declared and the number of sequence of procedure calls. Methods of estimating the amount of stack required for a program are included in a later section of this manual.

The run command allows the amount of stack space to be specified on the command line. In the run command, the size of the stack is selected by following the program name with the stack size, separated with a blank or a comma. For example, the following line would cause the program DATABASE to execute with 15K (15360 bytes) of stack space. (No angle brackets around 15K required in run command)

```
RUN DATABASE 15K
```

The stack size can be specified as a decimal or hexadecimal number. Hexadecimal numbers have a '#' or '>' as the first character. This is the same notation as is used in the pascal language. The letter 'K' means 1024, so 8K is equivalent to 8*1024 or 8192. If no stack size is specified, then one half of the unused memory space is allocated for the stack, and the other half to the heap. The heap is the area of memory used by the pascal program for dynamic memory storage as required by the procedures NEW and DISPOSE.

When execution of the program completes, the amount of stack and heap used is displayed on the screen. These numbers reflect the actual quantity of memory used during execution.

THE PASCAL COMPILER LISTING

The pascal compiler reads the source program from a file and produces two outputs. One of these is a file containing the object code. This code is loaded when the program is executed. The other output of the compiler is the listing. The listing contains the text of the source program with some additional information.

The listing is divided into pages. At the top of each page is a heading. The heading contains the version number of the compiler, the time and date when the compile started, and the page number. Each page after the first contains a form feed (control/L or #0C) character. The form feed will cause a page eject on most printers.

Each line on the listing has the address of the code being generated on the left. This address is expressed in hexadecimal and is relative to the beginning of the procedure. Each procedure starts over at zero. These numbers may be used during debugging to identify the location where an error occurred. When a runtime error is detected, the runtime system displays the location of the error on the screen.

The starting address of each procedure in memory may be displayed by the "S" command within the linking loader. If starting address of the procedure in which the error occurred is subtracted from the address of the error, the resulting displacement may be used to identify the line within the source program where the error occurred.

Using Alcor Pascal on the TRS80

If errors are detected by the Pascal compiler, error messages will appear in the listing. Error message lines have a string of five asterisks ('*****') at the beginning of the line. An up arrow will appear pointing at the approximate location within the line where the error was detected. This will be followed by one or more error codes. It is possible for a single error to generate more than one error code. For example, a procedure argument which is an undefined variable also does not match the type of the parameter. In most cases the first error code identifies the cause of the error.

If any errors were detected, a summary of the meanings of the error codes that were generated is printed at the end of the listing.

USING THE ALCOR PASCAL LINKING LOADER

This section describes the use of the Pascal linking loader. The linking loader provides powerful facilities for configuring Pascal programs. Separately compiled programs and procedures may be linked together and executed. Programs may be linked and stored as command files on disk and then later invoked from TRSDOS as commands. These command files behave in the same way as the utilities supplied with the operating system. This section assumes that the reader is familiar with the Alcor Pascal reference manual and has some experience with Pascal on the TRS80.

Invoking the loader:

The loader is executed by typing "LINKLOAD" at the TRSDOS command level. At this point the linking loader is brought into memory from disk. The first item displayed is a menu of commands followed by the command prompt:

```
L=LOAD, R=RUN, T=TRSDOS, I=INIT, S=SYMBOLS, B=BUILD CMD
>>
```

Each of these commands will be described in detail later. All commands require only single letter, although longer names will also be accepted. A command is terminated with the ENTER key. To invoke a command, simply type its first letter followed by ENTER. If more information is required, additional prompts will be supplied. The list of commands can be displayed by typing "H" or "?".

Using Alcor Pascal on the TRS80

Loading Programs:

The load command is used to load programs, procedures and functions into memory. To load a program, type "L" and press the "ENTER" key. The load command will ask for a file name. Type the name of the file in standard TRSDOS notation. The file should contain object code as generated from the Pascal compiler. The object file will be opened, and the object code will be loaded into memory. Each time a program, procedure or function is loaded, its name will be displayed on the screen. This will allow you to monitor the load process, and shows the identity of the procedures being loaded.

The object code for each Pascal procedure is compiled into a separate entity. These are then linked together when they are loaded. This allows procedures to be compiled separately and then joined. Thus, a program may be compiled a piece at a time, and when changes are made, only the parts affected by the change need to be recompiled. This also allows the creation of libraries of utilities. These utilities can be loaded with any program that needs them, but need be compiled only once.

Symbols:

The linking loader records the name and address of each procedure in a table as it is loaded. Also in this table are the names of procedures that have been called (referenced) by another procedure, but have not yet been loaded into memory. This symbol table can be displayed to the screen with the "S" command.

The symbols command displays all currently defined or referenced symbols on the screen. One procedure name is displayed per line. After the procedure name is a character that describes the use of that procedure. A "D" indicates that the name is defined; that is, the procedure has been loaded into memory. An "R" indicates that the procedure has been referenced but not yet defined. This means that a procedure that has already been loaded makes a call to this procedure. All procedures that are called must be loaded before the program can run. A "C" indicates that the symbol is the name of a common block. Commons are used to provide statically allocated shared data. See the Alcor Pascal reference manual for an explanation of the use of commons.

The last item on the line is the address of the symbol. If the symbol is defined ("D") then this is the address in memory where the procedure begins. If the symbol has not been defined ("R") then this is the address of the last place it was used (called).

(C) copyright 1981 Alcor Systems

Using Alcor Pascal on the TRS80

Running Programs:

After a program has been loaded, it can be executed with the Run command. This command will prompt for two pieces of information. The first prompt asks for the name of the program. More than one program can be loaded at a time so you have the option of selecting which one to execute. Simply type the name of the program in response to the prompt. If the line is left blank (press enter only), then the most recently loaded program is run. This is the last name that was displayed during the LOAD command. If only one pascal object file is loaded, the run command will always execute the main program in response to a null entry.

The second prompt from the run command asks for the amount of stack space required by the program. As in the RUN program, one-half of the unused memory is allocated to stack, and the other half to the heap by default. If these space allocations are sufficient, then simply press the enter key. Otherwise enter a value. The size of the stack may be expressed in decimal, hexadecimal (precede the number with ">" or "#"), or in kilobytes. 8k means 8 times 1024, or 8192 bytes. Methods of estimating the required stack size are included in a later section of this manual.

The program will execute after the second prompt is answered. If files are used by the program, the names of the files to be used will be determined from the keyboard. When a file is opened with RESET or REWRITE, the pascal file name will be displayed on the screen and you will be requested to type the name of the actual file to be used. The names are in standard TRSDOS notation. If you wish to use an Input or Output device instead of a file, this can be specified in a manner analogous to disk names. A device is designated by a colon followed by a letter indicating the device. For example :L is the line printer and :C is the crt and keyboard.

Building command files:

Once a program has been loaded, it may be saved on disk as a command file. This is done by the build (B) command. The first two prompts from this command are the same as for the run command and have the same meaning. The build command then asks for a file name. This is the name of the file that will contain the generated command. The program will be saved on disk in TRSDOS loader format and may be run at a later time by typing its name to TRSDOS. The build command then returns control to TRSDOS. The program may be run by typing its name to TRSDOS.

Using Alcor Pascal on the TRS80

Init:

The init command clears the symbol table and redisplays the command menu. This command may be used if the wrong program is loaded by mistake. It is equivalent to exiting to TRSDOS and then running LOAD again.

Trsdos:

The T command returns to the TRSDOS operating system.

ERROR MESSAGES

The following error messages are generated by the linking loader:

*** CANNOT OPEN FILE

This message is generated when you attempt to load and the loader is unable to find a file by the name specified. This may be caused by a misspelling, or the wrong disk being in the drive.

*** UNRESOLVED REFERENCES

When you use the run command to execute a program, or the build command to generate an image on disk, the loader checks that all of the procedures that are called within the program have been loaded. If there are procedures or functions that have been called but have not been defined, then this message is generated. At this point, you can load the required modules and repeat the command. The symbols command can be used to list names of the procedures that are not yet defined. These will have an "R" in the listing.

*** INVALID OBJECT TAG

This message is issued when a load is performed on a file that is not in a valid object format. The most frequent cause of this error is an attempt to load the source program instead of the object.

*** SYMBOL TABLE FULL

The linking loader has room for 256 different external symbols. If more procedures than this are loaded, the symbol table will become full.

*** ILLEGAL REFERENCE

This message signifies an inconsistent structure in an object file. It is an indication that the file has been damaged. The best solution is to recompile the offending program.

Estimating Stack Size

Pascal programs use a stack to store local variables and to save return addresses for procedure and function calls. This stack is allocated when the program is run and the required size is determined by the number and type of variables declared and the number of and sequence of procedure calls. The stack is a dynamic structure. Space is allocated when a procedure is called and released when the procedure is exited.

The total stack size required by a program is determined from its dynamic behavior at run time. Each time a procedure is called, space is allocated for its local variables. The total stack in use is a function of the number of procedures active at the time and the number and sizes of variables used within those procedures. If two procedures are never active at the same time, then the space used by each can be shared. The total stack that must be allocated is determined from the maximum size that is in use at any given time.

The simplest method of determining stack requirements is to run the program. Specify enough stack for it to run, perhaps with an excess. When the program terminates, the maximum stack used by the program is printed on the CRT. A good rule of thumb is to allocate 20% more stack than is required for a typical execution of the program.

The size of stack required can also be determined from the source program. It is necessary to determine which procedures will be active at a given time. Then add the size of the local variables for each procedure. If too much or too little stack is allocated for the program, it may terminate unpredictably.

The sizes of simple variables are summarized below:

type	size in bytes
CHAR	1
BOOLEAN	1
INTEGER	2
STRING	2
REAL	4
REAL (double precision)	8
FILE	32
TEXT	32

Using Alcor Pascal on the TRS80

The size of an array is determined by multiplying the size of the array (upper bound-lower bound+1) by the size of an element. The size of a record is determined by adding the sizes of its individual fields. Packing is on byte boundaries.

The size of a set is one plus the ordinal of its largest possible member divided by 8. Enumerated types require one byte, and subranges are one byte if the upper bound is within 255 of the lower bound and two bytes otherwise. (0..255 requires one byte).

To calculate the total stack size required, you should also include 64 bytes for the predeclared files INPUT and OUTPUT. Active procedures require space for their parameters as well as their local variables. Parameters passed by value require storage based on the size of the variable; parameters passed by reference require two bytes each. Each active procedure also requires 9 bytes to store dynamic return information.

PASCAL MEMORY USAGE

The pascal linking loader or RUN program is loaded by TRSDOS at #5200 in memory. The pascal program that is being executed will be loaded immediately above the loader. The next segment above the program is used to contain the pascal stack. The stack is used by pascal to contain the local variables declared in the VAR section of each program, procedure or function. It also contains return addresses and linkage information.

The remainder of available memory is used for the heap. The heap is a section of memory that is used for allocating dynamic storage. Programs that use pointers and the procedure NEW, will use storage from the heap. The heap also contains the buffers used to read from and write to files.

The total amount of memory available to pascal is determined from a TRSDOS system constant. On the TRS80 model I, the location of the top of memory is stored at #4049 in system RAM. On the model III, the location is #4411. If other programs are to reside in memory along with pascal, they should be loaded at the top of memory. The top of memory address should be changed to prevent pascal from using the reserved locations.

Compiler memory constraints

The Alcor Pascal compiler requires approximately 33k of memory for code. Of this total, 27k is the compiler itself and the remainder is runtime support. The runtime support portion contains the drivers for input and output devices, an interface to the file system and the Pcode interpreter. TRSDOS occupies 4.5k of memory, which leaves 10.5k bytes of memory for data. 4.5k of this total is used for stack space by the compiler, with the result that the heap is approximately 6k bytes. This is enough space for about 250 symbols to be defined. A program that uses more than 250 symbols at a time will run out of heap space during the compile.

There are some ways of saving memory during the compile so that larger programs can be compiled. The limit on symbols is relative to the number of symbols visible at any point within the program. Symbols that are not available to the program are not retained by the compiler. The use of symbol table space can be improved by defining fewer global variables at the outer levels and making use of locals whenever possible. This is also good programming practice.

The length of symbol names is not relevant in Pascal, unlike BASIC. Use of long names has no effect on program size or compiler memory usage. Extensive use of string constants will cause the compiler to use more memory. If a string constant is used in more than one place in the program, it will take less space if it is declared as a constant.

PASCALB is the overlayed or segmented version of the compiler. This version dynamically loads portions of the Pascal compiler from disk as needed. This increases the amount of memory available for symbols and allows larger programs to be compiled. The overlayed compiler will compile programs that are four times the size that can be compiled with the non-overlayed compiler. I.E.; a typical 4000 line program will compile successfully. It also has the drawback that the compiler runs more slowly.

REAL NUMBERS

Real numbers are either single precision or double precision. The TRS80 rom routines are used for all floating point calculations, and the precision and accuracy of calculations are the same as for Basic programs. Whether real numbers in Alcor Pascal are considered to be single or double precision by the compiler is set by a compiler switch setting at compile time. See the Alcor Pascal Reference Manual , "compiler options".

Single precision

Mantissa	Exponent
6 digits	-38..+38

Double precision

Mantissa	Exponent
16 digits	-38..+38

TRS80 Procedure and Function Library

A set of functions and procedures to access the hardware features of the TRS80 is provided with the Alcor Systems pascal compiler. These procedures can be declared as external procedures within pascal programs. The object code for these procedures and functions is provided in two forms.

If the program is executed with the RUN command, the function library is contained within the RUN program. Any of the library procedures and functions can be called and the routine will be linked to when the program is loaded. If the linking loader is used, these routines are not automatically available. This allows programs that do not need these routines to have more space available. The function library is provided in object form on disk. This file can be loaded with the load command from the linking loader. This will make all of the library routines available.

Each of the library routines is described below. A pascal external declaration is given. This declaration should be included in any program that uses the routine.

The external declarations of the library routines are included in a file on the release disk. Any or all of these declarations can be inserted into the source program using the insert file command in the text editor.

PROCEDURE CLEARGRAPHICS; EXTERNAL;

The purpose of this procedure is to clear the display when utilizing the graphics routines. Its function is similar to the clearscreen function but loads all hex 80´s into the display memory, instead of hex 20´s as CLEARSCREEN does.

PROCEDURE SETPOINT(X, Y : INTEGER); EXTERNAL;

This procedure sets a graphics point on the screen. The location of the point is specified with the x (horizontal) and y (vertical) coordinates. The value of x should be in the range: 0 <= x <= 127 The value of y should be in the range: 0 <= y <= 47

PROCEDURE RSETPOINT(X, Y : INTEGER); EXTERNAL;

This procedure clears a graphics point on the screen. The location of the point is specified with the x (horizontal) and y (vertical) coordinates. The value of x should be in the range: 0 <= x <= 127 The value of y should be in the range: 0 <= y <= 47

FUNCTION TESTPOINT(X, Y) : BOOLEAN; EXTERNAL;

This function tests the state of a point on the screen in graphics mode. X and Y are the horizontal and vertical coordinates of the point to be tested. The function returns TRUE if the point is on (white), and FALSE if the point is off.

```
TYPE
   BYTE = 0..255;
FUNCTION PEEK(ADDRESS : INTEGER) : BYTE; EXTERNAL;
```

This function returns the contents of any memory location. It may be used to examine memory or memory mapped input devices. ADDRESS is the address being examined. An address may be passed if its value is known. The addresses of pascal variables may be obtained by calling the LOCATION function. (see Reference Manual)

```
PROCEDURE POKE(ADDRESS : INTEGER; VALUE : BYTE); EXTERNAL;
```

Poke is used to alter the contents of any location in memory. It may also be used to write to memory mapped output devices such as the printer port.

```
PROCEDURE GOTOXY(X, Y : INTEGER); EXTERNAL;
```

This procedure positions the cursor on the CRT to the specified location. If a write is performed to a file connected to the screen then the text will appear beginning at the addressed location. The procedures WRITECH and WRITESTRING (see below) also use this location. The value of x should be in the range: 0 <= x <= 63 The value of y should be in the range: 0 <= y <= 15 If GOTOXY is used inconjunction with Pascal READ or WRITE statements, then a call to the external procedure NOBLANK at the beginning of the program is necessary. The TRS80 ROM driver for the screen will automatically clear the next line on the display when the carriage return character is received. The can be detrimental when constructing menu displays. A call to NOBLANK will cause the next line to always be re-displayed.

```
PROCEDURE NOBLANK(REDISPLAY : BOOLEAN); EXTERNAL;
```

The TRS80 ROM routine driver for the screen will automatically clear the next line on the display when a CR character is received. A call to NOBLANK with REDISPLAY := TRUE will cause the next line to always be preserved if REDISPLAY := false, it will be blanked. The Pascal logical files used for screen display must be RESET after the NOBLANK call for it to take effect. This includes INPUT.

```
PROCEDURE READCURSOR(VAR X, Y : INTEGER); EXTERNAL;
```

This procedure returns the current position of the cursor on the crt screen. X is the horizontal position (character) and Y is the vertical position (line).

```
PROCEDURE WRITECH(CH : CHAR); EXTERNAL;
```

This procedure writes a single character to the CRT at the current cursor location. The cursor location is advanced by one.

```
TYPE
  CHARSTRING = PACKED ARRAY[1..XX] OF CHAR;
  PROCEDURE WRITESTRING(VAR S : CHARSTRING; FIRST, LAST : INTEGER);
     EXTERNAL;     (XX is any length)
```

This procedure writes a portion of a string of characters to the screen. The text is written starting at the current cursor location. FIRST is the index of the first character to be written, LAST is the index of the last character to be written. The total number of characters displayed is: LAST-FIRST+1. If last is less than first then no characters are written.

PROCEDURE CLEARSCREEN; EXTERNAL;

A call to CLEARSCREEN causes the crt display to be cleared and the display to be set to 64 character width.

PROCEDURE INKEY(VAR CH : CHAR; VAR READY : BOOLEAN); EXTERNAL;

This procedure scans the keyboard to determine if a key is being pressed. If a key is currently pressed, then CH is the character generated by that key and READY is set to TRUE. If no key is pressed, then READY is FALSE and CH is the space character: ' '.

FUNCTION GETKEY : CHAR; EXTERNAL;

This function waits for and returns the next character from the keyboard.

FUNCTION INP(PORT : BYTE) : BYTE; EXTERNAL;

This function performs input from a Z80 IO port. The port number is passed to the function and the value read from that port is returned as the function value.

PROCEDURE OUT(PORT, VALUE : BYTE); EXTERNAL;

This procedure performs physical output to a Z80 port. It may be used in conjunction with the function INP to communicate with devices interfaced as input or output ports. The two parameters specify the port number and the value to be written to that port.

PROCEDURE USER(ADDRESS : INTEGER; VAR DATA : INTEGER); EXTERNAL;

This procedure interfaces to assembly language routines resident in the TRS80's memory. ADDRESS is the physical address where the routine is loaded. Any assembly language routines that are to be called from pascal should be loaded in a portion of memory that is not used by TRSDOS or PASCAL. The location of the top of memory can be set by using the TRSDOS model I debugger to alter location #4049 in RAM, or location #4411 on the model III. This location contains the highest usable location in memory. Pascal will not use any memory above this address, so assembly language routines can be loaded there.

Information is passed to the assembly language routine through the DATA parameter. When the assembly language routine is called, the HL register pair contains the value of DATA. When the routine exits, the contents of the HL register pair is returned as the new value of DATA. In cases where more than one word of information is required, the value of DATA can be the address of a variable. The address of any pascal variable can be obtained by a call to the predefined function: LOCATION. This enables the called assembly language routine to access arrays or buffer data areas. The assembly language routine is entered with a standard Z80 call instruction and should be exited via a return. All Z80 registers are available for use in the assembly language subroutine.

```
PROCEDURE CALL$(ADDRESS : INTEGER; VAR A : BYTE;
 VAR BC, DE, HL, IX, IY : INTEGER); EXTERNAL;
```

This procedure can be used in a similar manner to USER to call assembly language subroutines. The difference is that CALL$ permits you to set up all of the Z80 registers from pascal. The values passed will be in the registers when the subroutine is called. When the subroutine returns, the current contents of all registers are returned to the pascal program via the reference parameters.

```
TYPE
   ALPHA = PACKED ARRAY[1..8] OF CHAR;
   PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
```

This procedure returns the current time of day. The time is in the form of: hh:mm:ss

```
TYPE
   ALPHA = PACKED ARRAY[1..8] OF CHAR;
   PROCEDURE DATE(VAR T : ALPHA); EXTERNAL;
```

This procedure returns the current date as known to the operating system. The date is returned as: mm/dd/yy

TRS80 Procedure and Function Library

FUNCTION FILE$STATUS(VAR F : TEXT) : BYTE; EXTERNAL;

This function returns the status of a file. The file can be of any type, but the external declaration must specify a type that matches the type of file being tested. The byte returned is the error code for the latest IO (input or output) error. If no errors have occurred, then zero is returned. This function is used in conjunction with IO$ERROR and allows a program to detect and recover from its own IO errors.

PROCEDURE IO$ERROR(NEWSTATE : BOOLEAN;
VAR OLDSTATE : BOOLEAN); EXTERNAL;

This procedure sets the state of the IO error recovery flag within the pascal runtime system. This flag is used to determine whether a program detects its own IO errors. If the flag is set to true, then default error processing is performed. In case of an error on a file or device, a message is displayed on the CRT and the program halts.

If the IO error flag is set to false, then all IO errors are ignored by the system, and it is up to the program to check for and recover from IO errors. IO errors can be detected by calling the function FILE$STATUS.

NEWSTATE is a boolean value that sets the new state of the IO error recovery flag. OLDSTATE is used to return the previous value of the flag. This allows a program to change the state temporarily and then restore it.

PROCEDURE HP$ERROR(NEWSTATE : BOOLEAN;
VAR OLDSTATE : BOOLEAN); EXTERNAL;

This procedure sets the state of the heap error recovery flag within the pascal runtime system. When this flag is set to true, then a call to the procedure NEW will cause the program to terminate with an error message if no more space is available. Setting this flag to false causes the procedure NEW to return NIL if no space is available. The calling program should check for NIL on each call to NEW when this flag is set to false. This allows a program to use maximum memory from the heap without danger of an abnormal termination when space is exhausted.

```
TYPE
   FILENM = PACKED ARRAY[1..XX] OF CHAR;
   ALPHA  = PACKED ARRAY[1..8] OF CHAR;
(Where XX is any length long enough for the filename)

PROCEDURE SET$ACNM(VAR F : TEXT; VAR file name : FILENM;
   NAMELENGTH : INTEGER; VAR FILEID : ALPHA); EXTERNAL;
```

SET$ACNM is used to set the name of the physical file or device to be associated with a pascal file. It allows a program to compute file names internally. For example, a database program may know the name of the file containing the database. This procedure allows the program to specify the file name rather than requesting it from the keyboard.

The parameter F can be a file of any type. The external declaration of SET$ACNM that is included in the source program must specify a type for F that matches the actual file type to be used.

File name is a string containing the text of the file name. This string must be compatible with the operating system syntax for file names. The physical devices: lineprinter (:L), crt (:C) and dummy (:D) may also be used. NAMELENGTH is an integer that specifies the length of the file name.

FILEID is an 8 character string that is used to identify the Pascal name for the file, such as INPUT or OUTPUT.

If SET$ACNM is called prior to a RESET or REWRITE on a file, then Pascal will not prompt the CRT for the file name. All subsequent RESET or REWRITES will not cause a prompt unless a CLOSE(file name) is performed on the file. The file name association will remain as previously defined by SET$ACNM.

(Example program segment)

```
 TYPE
 FILENAME = PACKED ARRAY [1..15]OF CHAR;
 ALPHA    = PACKED ARRAY [1..8]OF CHAR;
 VAR  FNAME :FILENAME;
      FILEID:ALPHA;
      F     :TEXT;
PROCEDURE SET$ACNM(VAR F:TEXT; VAR FNAME:FILENAME; LEN:INTEGER;
                   VAR FILEID:ALPHA); EXTERNAL;
 BEGIN
    (* THIS ASSIGNMENT STATEMENT REQUIRES THE NAME TO BE LEFT *)
    (* JUSTIFIED, AND BLANK PADDED TO THE CORRECT ARRAY LENGTH *)
    FNAME:='DATA/TXT:0     ';
    FILEID:='F       ';
    SET$ACNM(F,FNAME,10,FILEID);
    RESET(F);
    READ(F,CH);
    (* AND ETC....... *)
```

PROCEDURE $MEMORY(VAR STACK, HEAP : INTEGER); EXTERNAL;

This procedure allows a program to determine the amount of memory currently available. The parameter STACK returns the current number of stack bytes available and the parameter HEAP returns the amount of heap available.

STRING FUNCTION LIBRARY

The following functions are provided for handling string manipulations. (See reference manual for additional information)

FUNCTION LEN(S : STRING) : INTEGER;
This function returns the length of a string.

FUNCTION LEFT$(S : STRING; POSITION : INTEGER) : STRING;
This function returns the left portion of the string ending at the specified position within the string.

FUNCTION RIGHT$(S : STRING; POSITION : INTEGER) : STRING;
This function returns the right portion of the string starting at the specified position within the string.

FUNCTION MID$(S : STRING; POSITION, LENGTH : INTEGER) : STRING;
This function returns the portion of the string starting at the specified position and including the number of characters specified by length.

FUNCTION STR$(LENGTH : INTEGER; CH : CHAR) : STRING;
This function returns a string of the specified length which is filled with the specified character.

FUNCTION ENCODEI(N : INTEGER) : STRING;
This function returns a string which is the character representation of the specified integer.

FUNCTION ENCODER(R : REAL) : STRING;
This function returns a string which is the character representation of the specified real. For single precision.

FUNCTION ENCODED(R : REAL) : STRING;
Same as ENCODER, but for double precision reals.

FUNCTION DECODEI(S : STRING) : INTEGER;
This function returns an integer number which is the binary representation of the specified string.

FUNCTION DECODER(S : STRING) : REAL;
This function returns a real number which is the binary representation of the specified string. For single precision.

FUNCTION DECODED(S : STRING) : REAL;
Same as DECODER, but for double precision reals.

```
FUNCTION CHARACTER(S : STRING; POSITION : INTEGER) : CHAR;
   This function returns the character at the specified
   position in the string.

TYPE COMPAREVALUE = (LESS, EQUAL, GREATER);
FUNCTION CMPSTR(S1, S2 : STRING) : COMPAREVALUE;
   This function compares the two specified strings
   and returns an enumerated value based on the
   comparison.  The returned value is LESS if S1<S2,
   EQUAL if S1=S2, and GREATER if S1>S2.

FUNCTION CONC(S1, S2 : STRING) : STRING;
   This function returns a string which is the result of
   the concatenation of the two specified strings.

FUNCTION CPYSTR(S : STRING) : STRING;
   This function returns a copy of the specified string.
   The typical use for this function is in the assignment
   of one string variable to another.  This prevents both
   string variables from referencing the same string.  EG.
   STRING1:=CPYSTR(STRING2);   will cause STRING1 to refer
   to a different copy of STRING2.  STRING1:=STRING2;  causes
   STRING1 to refer to the same copy of STRING2 and any changes
   in the value of STRING1 would cause STRING2 to change also.

FUNCTION DELETE(S : STRING; POSITION, LENGTH : INTEGER) : STRING;
   This function returns the string which results after deleting
   a specified number of characters beginning at the specified
   position.

FUNCTION FIND(SUBSTRING, S : STRING) : INTEGER;
   This function returns an integer number which points to the
   start of the specified substring within the specified string.
   If the string does not contain the substring then the returned
   value is 0.

FUNCTION INSERT(SUBSTRING, S : STRING; POSITION : INTEGER) : STRING;
   This function returns a string which is the result of inserting
   the specified substring into the specified string at the
   specified position.

FUNCTION REPLACE(OLDSTRING, NEWSTRING , S : STRING) : STRING;
   This function returns the string which results after
   replacing the old substring with a new substring.
```

Random Access Files

Random Access files refers to a file access method where any record may be READ or WRITTEN to in any order. As most Pascal programmers know, Pascal does not define the Random file type.

The following Pascal procedures and functions will allow random access to files on the TRS80. The following Pascal routines are supplied in object code format on the release disk.(RANDOM/OBJ) When using random access files, these routines should be declared as external in the main program. Then simply link to the supplied object file of random access routines with the linking loader to satisfy any external references.

All SOURCE and compiled object files for random access files are on the patch disk available to update 1.2 owners to 1.2A. Registered owners may send $8.00 + shipping to Alcor for this diskette.

The following declarations should be included in the source program.

RANDOM FILE ROUTINES

```
PROCEDURE OPENRAND(VAR F:FILETYPE; RECORDLEN:INTEGER;  PATHNAME:STRING;
                   VAR STATUS:INTEGER); EXTERNAL;
```

The purpose of this routine is to open a random file. The F variable is of any file type. Random file types are fixed in length and should be declared as a FILE OF DATATYPE. A text file is not a particularly useful DATATYPE. The filetype may be any structure such as an ARRAY, RECORD, etc... RECORDLEN must be the size required for the filetype. The SIZE(J) fucntion may be used to determine the RECORDLEN. PATHNAME is the physical

filename on disk. You must prompt the user if it is to be changed at runtime. STATUS is a code returned by PASCAL and the operating system. The status code returns the status of an operation on a random file.

```
PROCEDURE READRAND(VAR F:FILETYPE; RECORDNUM:INTEGER;
          VAR DAT:DATATYPE; VAR STATUS:INTEGER); EXTERNAL;
```

This routine is used to READ data from a random file. The RECORDNUM is the record number to be read. DAT is the buffer for the data and is declared to be of the same type the file declared DATATYPE in the OPENRAND routine.

```
PROCEDURE WRITERAND(VAR F:FILETYPE; RECORDNUM:INTEGER;
          VAR DAT:DATATYPE; VAR STATUS:INTEGER); EXTERNAL;
```

This routine is used to WRITE data to a random file. The RECORDNUM is the record number to be written. DAT is the buffer for the data and is declared to be of the same type the file declared DATATYPE in the OPENRAND routine.

```
PROCEDURE CLOSERAND(VAR F:FILETYPE); EXTERNAL;
```

Random files on TRSDOS are required to be closed before program termination. Failure to do so may result in a loss of data.

As with random files on any operating system, there are some peculiarities about random files. For example:

(1) If you WRITE record number 1 and WRITE record number 100, and then read any record from 2 to 99, the returned buffer will contain trash. The data will be whatever was previously on the diskette, probably the contents of an old file. This is because the operating system does not keep that much context. It is up to the user to keep track of unwritten records so they are not READ.

(2) Random file record sizes may be from 1 to 256 only. All blocking is taken care of by the system.

(3) The standard functions EOLN, EOF have no meaning for random files. The status codes as returned by the above routines perform those fuctions where applicable.

(4) The procedure OPENRAND is used to open a file for reading and writing to. Opening an empty file and reading is perfectly legal. It is up to the user to check the returned status on all random file operations.

(5) Random file record numbers are defined from 0..32,767 .

(6) As with normal files, if a file is declared LOCALLY within a procedure and opened, (Not passed in as a parameter) once the procedure is exited, Pascal will automatically close the file using the standard CLOSE file routine for non random files and postion the EOF mark in the directory at the last record read or written to. This may not be the correct position as desired by the programmer. An explicit call to CLOSERAND should be used to close the random file and position the EOF. This will always correctly place the EOF mark.

(7) You may declare a file to be:

```
(*WHERE XX IS ANY RECORD LENGTH FROM 1 TO 256*)
TYPE  LINE = ARRAY(.1..XX.) OF CHAR;
VAR   F:FILE OF LINE;
```

Once the file has been opened, you may access it by using the READRAND and WRITERAND external procedures even if the file was not created by Pascal. There is only one procedure for opening random files. (no reset and rewrite) You may read or write to a random file.

Random File Error Codes
Returned By External Procedures

15 - DISK WRITE PROTECTED
24 - FILE NOT FOUND
27 - DISK FULL
28 - END OF FILE
29 - RECORD NOT FOUND (PAST EOF)
128 - PATH NAME IS NULL OR TOO LONG
129 - RECORD LENGTH IS NOT BETWEEN 1 AND 256
130 - FILE IS ALREADY OPEN
131 - FILE IS NOT OPEN

(C) Copyright ALCOR SYSTEMS 1982

Pascal TUTORIAL
Alcor SYSTEMS

Second Edition
First printing
1982

Copyright (C) 1981 by Alcor Systems

SECOND EDITION
FIRST PRINTING-1982

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Alcor Systems. No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this book, Alcor Systems assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

TRS80 and TRSDOS are trademarks of Tandy Corporation.
USCD Pascal is a trademark of UC Regents, San Diego campus.
Apple is a registered trademark of Apple Computer Inc.
Z-80 is a trademark of Zilog corporation.
CPM is a trademark of Digital Research.

Preface

This book is intended to be a tutorial for Pascal programming. It was specifically designed as a learning aid for Alcor Pascal, and is an intermediate level tutorial guide. It is assumed that the reader has had some programming experience. This tutorial is an excellent teaching aid for most other Pascals because Alcor Pascal is an implementation of standard Pascal. Any extensions to the language are covered in the Alcor Pascal Reference Manual. In this book the standard Pascal referred to is defined by Pascal USER MANUAL AND REPORT(2nd edition) by Kathleen Jensen and Nikalus Wirth (Springer-Verlag, 1975). People with some exposure to basic or other programming languages should have no trouble understanding the explanations or example programs. It may be helpful to refer to the Alcor Pascal reference manual, for additional details and answers. This tutorial was designed to be as clear and precise as possible for the newcomer to Pascal. It avoids all tricky and confusing explanations, and in many cases includes program segments as examples. This greatly reduces the clutter that often gets in the way of learning computer languages.

The first chapter examines the major advantages of Pascal as a general programming language. You may skip this section and begin reading at chapter two if you wish. However, there are many important aspects about Pascal that are explained in chapter one. This tutorial will provide a logical and structured approach to learning . After all, that's what Pascal is all about.

(C) copyright 1981 Alcor Systems

Table of Contents

(C) copyright 1981 Alcor Systems

Table of contents

Chapter

(C) copyright 1981 Alcor Systems

INTRODUCTION

Pascal was created by Professor Nicklaus Wirth at the Swiss Technical Institute in Zurich Switzerland. It was first announced in 1965 when the most popular programming languages in use by the computer industry were Fortran and Cobol. In teaching environments, like Universities, Algol was a popular language for introducing students to computer programming. Wirth felt that languages like Fortran and Cobol were too loosely structured to promote good programming habits to students. Algol, although more structured, had significant drawbacks. Wirth decided to depart from normal teaching practice and designed a new language patterned after Algol, to be his new teaching language.

Pascal inherits the structured control statements of Algol and adds powerful data structuring capability. The language was designed to promote good programming practices and encourage clarity and modularity in programs. Since the first implementation of Pascal on the CDC-6600 computer system in 1971, Pascal has proven to be one of the most popular programming languages in existence.

Pascal has the distinction of being created for the purpose of making the development of computer programs a structured and logical process. Pascal contains the best features of most high level programming languages. Many college instructors at major universities today use Pascal or Pascal like languages to teach structured programming classes. Structured programming classes emphasize the use of guidelines and rules for developing computer programs. Some of the goals of structured programming are to encourage modularity and functionality, promote good documentation and to generate programs that have smooth flows of logic from the beginning to end. Programs are usually developed in Pascal or an English like Pascal and then hand translated to any available computer language such as Basic, for execution.

Although the implementation language may not be highly structured, the final program will be more clear and readable. Indeed, that is exactly what most Pascal programmers do when they need to use other languages. However, this is no replacement for implementing the program in Pascal, as there are no translations for the rich and powerful data structures and many other features that exist in Pascal.

(C) copyright 1981 Alcor Systems

Data types and structures are two important features of the language. They comprise one of the largest differences between languages such as Pascal and Basic. Most Basic programmers are familiar with the data types integer and real. A data type is simply the kind of information that may be stored in a variable. Pascal includes nine predefined types: char, integer, real, set, file,array,record, boolean and text plus an infinite variety more, as you may invent data types at will.

Data structure is another name for a variable type such as the array. Pascal allows you to build new data structures as desired. The use of record data structures can be very powerful when building or maintaining data bases. With one simple output statement, an entire data structure may be written to a file.

Variables are assigned storage only as needed during program execution, thus reducing demands on memory. They also may have names with as many characters in them as desired provided that the first 8 characters form a unique name. Long names don´t require any more storage space than short ones.

Extra spaces,tabs, and carriage control may be placed freely in a source program, except in the middle of identifiers and character strings. An identifier is defined to be a program, variable, constant, type, procedure or function name. Comments may be inserted anywhere spaces are allowed and are delimited by (* *) or { } . These features don´t affect the speed or the size of the final program, and greatly improve readability.

The concept of local variables is important. Variables declared in this manner will have restricted access by other parts of the program. This can prevent accidental changes in their values.

If there are a series of statements that need to be executed by different sections of the program, they may be placed in a procedure or function declaration. A procedure or function is just a collection of program statements that may be called to perform their task at various times during the program. Repetitive programming may be prevented by creating libraries of commonly used procedures or functions. Parameters may be passed to these subroutines by "value or reference".

(C) copyright 1981 Alcor Systems

When a parameter is passed by reference, the actual parameter is passed to the procedure, and if the procedure alters its value, the parameter's value is changed in the rest of the program. When a parameter is passed by reference , the argument must be a variable.

When a parameter is passed by value, what is passed is a copy of the argument. If the procedure alters the parameter's value, the value in the rest of the program is not changed. When a parameter is passed by value, it's argument may be a variable or any legal arithmetic expression. Parameters passed by value can prevent accidental changes in a value by procedures.

A careful use of procedures and functions will make the program more readable and will eliminate branching statements that are difficult to follow.

The logical operators AND , OR, and NOT along with the relational operators: greater than ">", less than "<", equal "=", not equal "<>", greater than or equal ">=", less than or equal "<=" are available in Pascal. Statements like : IF(count < 10) and (not FAILURE) then "do the following", make control statements very clear.

There are six statements in Pascal used for the flow of control. Loop control is performed by the FOR, REPEAT and WHILE statements. Conditions are tested with the IF and CASE statements. Branching is accomplished by the GOTO statement.

Program execution speed may be of particular importance in certain applications. Alcor Pascal programs execute between 10 and 50 times faster than most interpreted Basics on the same computers. In fact, they are significantly faster than many other Pascal implementations. Actual benchmark test results comparing Alcor Pascal with other language implementations may be found in the Alcor Pascal System Implementation Manual.

(C) copyright 1981 Alcor Systems

As a general programming language, Pascal has the following advantages; It includes:

(1) The powerful ability to build new data types and structures as desired.
(2) The control statements while, repeat, for, if, case and goto.
(3) The logical operators AND, OR, NOT.
(4) The relational operators: equal to, less than, greater than, less than or equal to, greater than or equal, not equal to.
(5) Recursive procedures and functions with parameter lists.
(6) The ability to insert blanks and comments in the source program easily,and long variable names, with no space or time penalty.
(7) User controlled dynamic memory management.
(8) Efficient memory management of variables,functions and procedures.
(9) Arrays of one or more dimensions.
(10) Record data structures.
(11) Sets and set operations.
(12) Subrange and enumerated data types.
(13) Named constants.
(14) Read and write statements plus formatted write statements.
(15) Built in functions and procedures.

Alcor Pascal has the added advantage of being a full implementation of standard Pascal, thus program portability is greatly enhanced. These features, and the fact that programs generated by Alcor Pascal execute much faster than programs generated by most Basic or other Pascal systems, make Alcor Pascal a logical choice as a general high level programming language.

(C) copyright 1981 Alcor Systems

STARTING CONCEPTS

At the simplest level of structure of a Pascal program are the program, begin, and end statements. They may be thought of as the outer shell that must be around all programs. The actual program is placed between these begin and end statements. Example:

Listing 1.1

```
PROGRAM test;
BEGIN
END.
```

This is a completely legal Pascal program although it actually does nothing. We can modify it by adding a writeln statement to it.

Listing 1.2

```
PROGRAM test;
BEGIN
  WRITELN(OUTPUT,'* Pascal is a very structured language.');
  WRITELN(OUTPUT,'* It promotes good programming habits.');
END.
```

The program will write to the the file associated with OUTPUT the following message.

```
* Pascal is a very structured language.
* It promotes good programming habits.
```

(C) copyright 1981 Alcor Systems

The two writeln statements comprise the only action in the program. The OUTPUT in the writeln tells the computer to write the message to the file associated with the logical name OUTPUT. How this association is accomplished is a computer dependent process, and is explained in the System Implementation Manual. The string in single quotes is a text string that may be composed of printable characters. Notice two things about this program. First, the text string may not be broken up across line boundaries, however blanks may be used freely elsewhere to make the program more readable. Secondly, a semi-colon is required after each writeln statement. In fact, semi-colons are required after most Pascal program statements. For now, a good rule of thumb is to always include a semicolon after legal Pascal statements. The program name is test, but may be any identifier where the starting character is a letter. The "." must always occur after the last END statement in the program.

Another output statement similar to the writeln statement is the write statement. In the first sample program the two messages were written to different lines on the file. The writeln statement caused the file position pointer to reposition to the beginning of the next line after each message was written. The file position pointer is another name for the cursor when the file I/O is directed to the terminal. The write statement, does not reposition the cursor after the message has been written. Instead, the cursor remains at the end of the last message, and the next text will appear on the same line. The cursor represents the point on a line where text will appear from the next write statement.

Listing 2.1

```
PROGRAM test;
   BEGIN
   (* the purpose of this program is to give an example *)
   (* of how to use the WRITE and WRITELN procedures    *)
   WRITE(OUTPUT,' * Now is the time');
   WRITE(OUTPUT,' for all good programmers');
   WRITE(OUTPUT,' to learn');
   WRITELN(OUTPUT,' Pascal.');
   (* The next statement starts on a new line *)
   WRITELN(OUTPUT,' * You will become a Pascal magician.');
   END.
```

(C) copyright 1981 Alcor Systems

The following message will be written to output.

```
* Now is the time for all good programmers to learn Pascal.
* You will become a Pascal magician.
```

If you noticed, the text enclosed between the (* *) did not affect the program execution. They are simply comments by the programmer to help clarify the logic in the program. Comments may be especially helpful later when you have forgotten how the program functions. They may be inserted anywhere except in the middle of identifiers or text strings. An identifier is just another name for a program, variable, constant, procedure or function name. Procedures and functions will be explained later.

Tutorial Quiz 2.0

(1) The first statement of a Pascal program must be the _______ statement.

(2) The ________ statement will not move the cursor to beginning of the next line.

(3) The ________ statement will move the cursor to the beginning of the next line.

(4) Most Pascal statements are followed by a _________.

(5) The ____ statement must be the last statement of a program.

(6) Quoted _________ may not be broken up across line boundaries.

(C) copyright 1981 Alcor Systems

DATA CONCEPTS

Variables

Variables in Pascal serve the same purpose as they do in most other programming languages. They serve as storage areas for the information that the programmer may wish to manipulate. These storage areas are referred to by names that are chosen by the programmer. Each variable name must start with a letter. It may be composed of any combination of letters and digits, although in many Pascal implementations, the first eight characters must form a unique name within the program.

Reserved words

There are certain words in Pascal that have special meanings. These words are called reserved words, and variables may not have these names. For a complete list see the Alcor Pascal Language Reference Manual.

Variable types

Variables must have associated with them a specific type. The type is the kind of information that is going to be stored in that variable. For example, the variable "taxnumber" may represent a business tax number. This taxnumber might take on the numerical value of 1 to 100 at any time in the program. This would be an example of the type, integer.

Declaring variables

All variables must have their specific type declared in a special section of Pascal programs called the var section. There are five predefined variable types in Pascal that we will concern ourselves with at this time. They are integer, real, char, text and boolean. The var section of a program consists of the word VAR followed by any number of variable declarations. A variable declaration has the form of variable name: variable type; . A colon separates the variable name from the variable type, and a semicolon must follow each variable declaration.

(C) copyright 1981 Alcor Systems

Integer variables

The type integer may be used to represent whole numbers. The minimum and maximum size allowed by Pascal is computer dependent, but on many micro computers they range from -32768 to +32767 . The following is a program example of a variable declared as an integer. Notice that a colon is required to separate the variable name taxnumber, from the variable type, integer.

Listing 3.1

```
PROGRAM test;
VAR
taxnumber:INTEGER;
BEGIN
END.
```

Real variables

The type real may be used where a variable must store numbers that may have fractional or decimal values. The numbers 2.98 , 3.047 , 0.0009 , 0.009 and 37.0998 are all examples of real numbers. Real numbers must start with a digit and may contain a decimal point. If a decimal point is present, a digit must follow the decimal point. The numbers .009, 10. are illegal real numbers, as there is no digit before and after the decimal point. The size and precision of real numbers are computer dependent . Real variables may represent the dollar selling price of some product by a store, or an entry into your checkbook. They are declared as follows:

Listing 3.2

```
PROGRAM test;
VAR
  taxnumbr:INTEGER;
  cost    :REAL;
BEGIN
END.
```

Note that the indentation of the declaration section does not affect the execution of the program.

(C) copyright 1981 Alcor Systems

Char variables

If a variable is declared as a char type, then it may represent a single character such as the character ´A´. In Pascal, the characters may be composed of letters, digits and other special symbols. If a digit is to be referred to as a character instead of a number, it is enclosed in single quotes like the character string was in program listing 2.1. The only difference is that a char variable may only represent one character at a time.

Text variables

Variables declared to be of the type text are used to direct output or input information to files on disks, or to other devices. Text is predefined to be a special file of char.

Boolean variables

A variable declared as the type boolean may only have two values. They are true and false. This kind of variable is primarily used in flow control statements. Boolean variables are typically used in the WHILE, IF or REPEAT control statements. These statements will be covered in later chapters.

Const section

Often, specific variables will have fixed values during program execution. In this case you may declare these values as constants. In Pascal, they are declared in the CONST section. The const declaration section is placed between the program and the first begin statement of the program. Constants may have names like variables do. In fact their names should reflect their nature. Constants may be integers, real numbers or a text string. A text string constant is any character string enclosed between single quotes. A string constant generally may be used anywhere a packed array[1..n]of char variable may be used. This variable type will be explained later.

(C) copyright 1981 Alcor Systems

(1) _______ serve as storage areas for information that the programmer may wish to manipulate.

(2) Variable types are declared in the ___ section of the program.

(3) Five predefined type of variables in Pascal are ________ , _____ , ________ , _____ , ______ .

(4) The syntax of a variable declaration is :
var variable name : ______ .

(5) Variables declared as the type ______ may take on the value of letters, digits and other special symbols.

(6) A variable declared to be of type _______ is used to direct I/O to files.

(7) A value that is fixed in the program and will not change may be declared as a constant in the ________ section of the program.

(C) copyright 1981 Alcor Systems

ADVANCED I/O

Procedures rewrite, writeln.

Can you guess what this program will do if you run it ?

```
Listing 4.1

PROGRAM alpha;
CONST
  pie     = 3.141597;
  maxtax  = 2000;
   tstring = ´ I am a  Pascal Wizard´;
VAR
  out    :TEXT;
  max    :REAL;
  number:INTEGER;
BEGIN
   REWRITE(out);
   WRITELN(OUTPUT,´Program starting execution.´);
   WRITELN(´ The value pie = ´,pie);
   WRITELN(´ The value maxtax = ´,maxtax);
   WRITELN(tstring);
   WRITELN(out,´This program tests file I/O´);
   WRITELN(OUTPUT,´Program finished.´);
END.
```

From example 2.1 you already know that the first and last writeln statement will cause the program to direct the messages to the file associated with output. The following message will be written to output.

```
Program starting execution.
 The value pie = 3.14159
 The value maxtax =       2000
 I am a Pascal wizard
Program finished.
```

The message, "This program tests file I/O", will be written to the file associated with out.

Examine the first writeln statement. In the specific case where the first argument for the writeln statement is output, the user is not required to declare output in the var section as with other files. Notice also that there is no output argument in the second,third and fourth writeln statements. In Pascal, it is not required to have output as an argument. Output is a default argument. Ie; the statements writeln(output,´ help´); and writeln(´ help´); are equivalent in Pascal. In Pascal the write and writeln statements may have multiple arguments. The first argument always directs the I/O operation to a specific file except for the case previously explained. In listing 2.1 the two arguments were output and a text string. Constants and variables may also be arguments. The values of the variables and constants will be written in the same order as they appear in the argument list.

Rewrite statement

The purpose of the rewrite(logical filename) statement is to open a file on some hardware device, and ready it for writing. Note that the previous contents of any file used in a rewrite statement will be lost. The specifics of how to associate the logical filename in parentheses with a physical filename is implementation dependent and is explained in the Alcor Pascal System Implementation Manual. Standard Pascal does not require the file output to have a rewrite performed on it before it is written to. Output is the only file in Pascal that does not require a rewrite before it is written to. It is predeclared to be a textfile by Pascal.

Reset statement

The purpose of the reset statement is to ready a file for reading to a program. A reset (logical filename) statement will open the physical file associated with the logical filename and read the first line. In Alcor Pascal, the first line is not read until required by an EOF or EOLN function call. These functions will be explained later. All files that are to be used for reading must be reset, except Input. Input is a predeclared textfile within Pascal.

Read, readln statements

The read statement is similar to the write statement, except that it´s purpose is to read information into the program instead of to write information. The read statement will read a value into a variable from a file and will leave the cursor at the last character read.

(C) copyright 1981 Alcor Systems

Specific reads on the same file will cause a series of inputs to occur from the same line. When a read is performed on an integer or real quantity in a text file, the read will start scanning the line until any non-blank character is found. The next contiguous non-blank characters will be interpreted by the read as the input value. If another read is performed on the same file, the read procedure will scan forward and repeat the process, until the end of line is reached. If the end of line is reached before any integer is found, the scan will continue at the beginning of the next line.

The readln statement performs the same function as the read statement, except that the cursor will always be positioned to the beginning of the next line after all inputs to the read statement are satisfied, even if the end of line has not been reached. The readln statement is not required to have arguments. The effect of such a readln is to position the cursor to the beginning of the next line without reading any values. The arguments allowed for the read statement are variables. As with OUTPUT in the write statement, INPUT is predeclared to be a text file. If a read statement does not have a file argument, it is assumed to be the predeclared file INPUT.

Try running the following program. It will give you a little more experience performing program I/O.

Listing 4.2

```
PROGRAM testIO;
(* Author- Alcor Systems                                  *)
(* Purpose- the purpose of this program is to             *)
(* demonstrate I/O to a text file using integer and       *)
(* real input variables.                                  *)
VAR
   taxnumbr,emnumber    :INTEGER;
   tax                  :REAL;
   ID                   :PACKED ARRAY[1..72]OF CHAR;
BEGIN
   WRITELN(OUTPUT,'* Enter your federal tax number: ');
   READLN(INPUT,taxnumbr);
   WRITELN(OUTPUT,'* Enter your dollar tax total: ');
   READLN(INPUT,tax);
   WRITELN
      (OUTPUT,' * Enter your employee number,a space,');
   WRITELN(OUTPUT,' followed by your business ID number:');
   READ(INPUT,emnumber);
   READLN(INPUT,ID);
   WRITELN(OUTPUT,' Tax number        = ',taxnumbr);
   WRITELN(OUTPUT,' Dollar tax total = ',tax);
   WRITELN(OUTPUT,' Employee number  = ',emnumber);
   WRITELN(OUTPUT,' Business I.D.    = ',ID);
END.
```

(C) copyright 1981 Alcor Systems

The following I/O will occur at the terminal if the filename associated with input and output is the local terminal.

```
* Enter your federal tax number:
32000                                  <user input>
* Enter your tax total:
2345.98                                <user input>
* Enter your employee number,a space,
followed by your business ID number:
23455 4669                             <user input>
Tax number          = 32000
Dollar tax total    = 2345.98
Employee number     = 23455
Business I.D.      = 4669
```

Tutorial Quiz 4.0

(1) The predefined file variables _______ and _______ are not required to be declared in the var section as the type text.

(2) The first argument in a ______ , _______, ______, _______ statement directs I/O to a file or device.

(3) The purpose of the _______ statement is to open a file and ready it for writing.

(4) The purpose of the ______ statement is to open a file and ready it for reading.

(5) After a _______ , the previous contents of the file are lost.

(6) A ______ or ______ statement will change the cursor's position after execution.

(C) copyright 1981 Alcor Systems

STATEMENTS

Assignment statements

From previous examples, you know how to read a value into a variable and how to write it. Now you will learn how to alter its value within the program. The statement that does this is the assignment statement. It allows you to set a variable´s value equal to an expression. An expression may be a variable name or a series of arithmetic or boolean operations. A simple assignment statement takes the form of variablename1 := variablename2; . The " := " operator causes the variable on the left hand side to become equal to the value of the variable on the right hand side.

Listing 5.1

```
Program MAGIC;
VAR
   intrate,principle,anint,calc:REAL;
BEGIN
   WRITELN(´ ******* Interest rate problem *******´);
   WRITELN(´ Enter annual interest rate:´);
   READLN(intrate);
   WRITELN(´ Enter the principle amount of loan:´);
   READLN(principle);
   calc:= intrate * principle;
   anint:=calc;
   WRITELN(´ Your annual interest payment = ´,anint);
END.
```

Arithmetic operators

In the program listing 5.1 you may have noticed the statement "calc:= intrate * principle" . The " * " is the multiply operator in Pascal. There are seven arithmetic operators in Pascal with precedence as follows :

OPERATOR PRECEDENCE TABLE 5.1

Symbol	Precedence	Description
-	(1) Highest	Unary operator. Negates a single argument.
*	(2)	Multiplies two arguments
/	(2)	Divides two real arguments
div	(2)	Divides two integer arguments
mod	(2)	Divides two integer arguments and keeps the remainder as the result.
+	(3) Lowest	Adds two arguments
-	(3) Lowest	Subtracts two arguments

Operator precedence

If an arithmetic expression is composed using different operators without any parentheses, the order of evaluation is based on the above table, where operations with the highest precedence are performed first. Any operations at the same level are performed in left to right order.

Parentheses

In Pascal, this natural order of precedence may be altered by enclosing a portion of the expression in parentheses. The parentheses has the highest precedence of all operators. Parentheses may be nested to alter the evaluation sequence as desired. In this case, operations buried deepest within are evaluated first.

(C) copyright 1981 Alcor Systems

The following program will illustrate the use of the arithmetic operators and parentheses.

```
Listing 5.2

PROGRAM math;
CONST
   fudge    = 100;
   lossacre = 0.50;
VAR
   acsoy,acgreen    :INTEGER;
   prsoy,prgreen    :REAL;
   profit,overcost  :REAL;
BEGIN
   WRITELN(OUTPUT,' **** Farmers profit analysis program **** ');
   WRITELN(OUTPUT,'* Please enter the following information:');
   WRITELN(OUTPUT,'* Acres planted in soy beans = ');
   READLN(INPUT,acsoy);
   WRITELN(OUTPUT,'* Profit per acre of soybeans = ');
   READLN(INPUT,prsoy);
   WRITELN(OUTPUT,'* Acres planted in green beans = ');
   READLN(INPUT,acgreen);
   WRITELN(OUTPUT,'* Profit per acre of green beans = ');
   READLN(INPUT,prgreen);
   WRITELN(OUTPUT,'$$$ COMPUTATION IN PROGRESS $$$');
   profit:= acsoy * prsoy + acgreen * prgreen
            - (fudge / (acsoy+acgreen) * lossacre);
   WRITELN(OUTPUT, ' Your computed profit is ');
   WRITELN(OUTPUT,profit);
END.
```

The profit calculation uses parentheses to alter the normal operator precedence. If the normal precedence is followed, the calculation will yield the wrong result.

The order of evaluation without parentheses would be:

(1) acsoy and prsoy multiplied.
(2) acgreen and prgreen multiplied.
(3) fudge / acsoy
(4) acgreen * lossacre
(5) result1 + to result2
(6) result5 - result3
(7) result4 + result 6

The desired result is obtained by including the parentheses as in the example. The apparent order of evaluation would be:

```
profit := acsoy * prsoy + acgreen * prgreen
       -(fudge / (acsoy+acgreen) * lossacre) ;
```

(1) acsoy added to acgreen
(2) fudge / result1
(3) result2 * lossacre
(4) acsoy * prsoy
(5) acgreen * prgreen
(6) result4 + result5
(7) result6 - result3

If two numbers are operated on, the normal result will have a type that is dependent on the argument types. The variable types required to store the results of specific operations are summarized in the following table.

*	multiply	real * integer = real result. integer * real = real result. real * real = real result. integer * integer = integer result.
/	real divide	real / real = real result. real/integer = real result. integer/real = real result. integer/integer = real result

div	integer divide	integer div integer = integer result. integer arguments only.
mod		integer mod integer = integer (integer div integer= remainder)
+	add	integer + integer = integer result. integer + real = real result. real + integer = real result. real + real = real result.
-	subtract	integer - integer = integer result. integer - real = real result. real - integer = real result. real - real = real result.

Compound statements

If a series of program statements are surrounded by a begin and end statement, then the enclosed statements are considered a compound statement. Compound statements are normally used as arguments to control structures such as the WHILE and IF. A compound statement may occur by itself anywhere in a Pascal program, however, its meaning would be the same as if the begin and end were not present. The important thing to remember about Pascal is that anywhere a single statement may be used, a compound statement may be used.

(C) copyright 1981 Alcor Systems

(1) ____ is the symbol for the assignment operator.

(2) If a series of statements are surrounded by a begin and end , it is called a _________ statement.

(3) Operator precedence refers to the order in which an __________ is evaluated.

(4) The natural order of expression evaluation may be altered by using ____________ .

(5) ____________ with the highest precedence will be evaluated first.

(6) Operators that have the same level of precedence will be evaluated in _____ to ______ order.

(7) After executing the following Pascal statement, variable x will have the value ________.

```
PROGRAM QUIZ;
VAR
   x:integer
BEGIN
   x:=4 + 5 * 2;
END.
```

(C) copyright 1981 Alcor Systems

FLOW CONTROL

FOR statements

If you wish to execute a series of statements a predetermined number of times, you should use the FOR statement. The for statement will cause a single or compound statement to execute a specific number of times. Examine the following example .

Listing 6.1

```
PROGRAM math;
CONST
   fudge    = 100;
   lossacre = 0.50;
   prsoy    = 195.98;
   prgreen  = 200.56;
VAR
   acsoy,acgreen,nofields,select,fieldnumber:INTEGER;
   profit,overcost: REAL;
BEGIN
   WRITELN(OUTPUT,'* Farmers planting analysis program * ');
   WRITELN(OUTPUT,'* How many fields do you have ?');
   READLN(INPUT,nofields);
   FOR fieldnumber := 1 to nofields DO
      BEGIN
      WRITELN(OUTPUT,'* For field number ',fieldnumber);
      WRITELN(OUTPUT,'* Acres planted in soy beans = ');
      READLN(INPUT,acsoy);
      WRITELN(OUTPUT,'Acres planted in green beans = ');
      READLN(INPUT,acgreen);
      profit:= acsoy * prsoy + acgreen * prgreen
               - (fudge / (acsoy+acgreen) * lossacre);
      WRITE(OUTPUT, '* Your computed profit for field number  '
                ,fieldnumber,'   is ');
      WRITELN(OUTPUT,profit);
      END;
END.
```

The loop control variable is "fieldnumber" . This variable is declared as an integer. When the loop starts its execution, "fieldnumber" takes on the value of one for the first pass through the loop. Successive loop iterations cause this value to be incremented by one until its value is greater than "nofields" . At this point, the loop will stop and control will be passed to the next statement in the program. The lower and upper bounds on the loop control variable do not have to be variables or constants, but may be arithmetic expressions. The expression is evaluated one time, at the beginning of the loop. The upper bound must be greater than or equal to the lower bound for the loop to execute at least once.

A variation on the for loop just described causes the loop control variable to be decremented by one instead of incremented by one. The syntax for this is the same as above except that the "to" in the for statement is replaced with "downto" . The initial upper bound on the loop control variable must be larger than or equal to the lower bound for the loop to execute at least once.

Case statement

The case statement is used as a selection control statement. It is used when you need to execute one statement from a list of statements. Notice the following program. In front of every statement in the list, is a case selector constant. This selector value must be of the same type as the case selector variable, and may be composed of a list of values for each statement it precedes. The "end" must follow the last statement in the list in order to terminate a case statement. We will be concerned with selector variable of type integer at this time.

(C) copyright 1981 Alcor Systems

Listing 6.2

```
PROGRAM moonphase;
CONST
   dayphcorr  = 10;
   lencycle   = 28.3;
VAR
   daynumber,intphase                    :INTEGER;
   startphase,phase,month,day,year :INTEGER;
   realphase,phasecorrection         :REAL;
BEGIN
   WRITE(OUTPUT,' *** Lunar Phase calculation program');
   WRITELN(OUTPUT,' ***');
   WRITELN(OUTPUT,' Enter the month/day/year:');
   READLN(INPUT,month,day,year);
   startphase := ((year-78) * 365) + dayphcorr ;
   CASE month of
     1: daynumber:=1;
     2: daynumber:=32;
     3: daynumber:=60;
     4: daynumber:=91;
     5: daynumber:=121;
     6: daynumber:=152;
     7: daynumber:=182;
     8: daynumber:=213;
     9: daynumber:=243;
    10: daynumber:=274;
    11: daynumber:=304;
    12: daynumber:=334;
   END;     (*case*)
   startphase := startphase + daynumber + day;
   realphase := startphase / lencycle;
   intphase := TRUNC(realphase);
   realphase:=realphase-intphase;
   phase:=realphase * lencycle;
   CASE phase OF
   1,2,3,4,5,6,7         : WRITELN(OUTPUT,
                            'The moon is in its first quarter.');
   8,9,10,11,12,13,14    : WRITELN(OUTPUT,
                           'The moon is in its second quarter.');
   15,16,17,18,19,20,21 : WRITELN(OUTPUT,
                           'The moon is in its third quarter.');
   22,23,24,25,26,27,28 : WRITELN(OUTPUT,
                           'The moon is in its fourth quarter.');
   END;        (*case*)
END.           (*PROGRAM*)
```

(C) copyright 1981 Alcor Systems

The purpose of the program in listing 6.2 is to compute the phase of the moon. Several examples of case statements are used with differing case selector lists. The calculations are based on a known starting phase of the moon at some past day,and year. The initial startphase calculation yields the number of days since this known starting date as a function of the number of years, corrected for the starting phase of the moon. The remainder of the calculations simply adjust this value to yield the whole number of days since the known starting phase, then divide the resultant number of days by the lunar cycle length in days. This program does not consider the effect of leap years. Notice that mixed mode expressions consisting of real and integer arithmetic are used throughout the calculations. A careful study of the previous type result tables will verify their validity. Notice that the value of realphase is used as an argument for the TRUNC function. This is a predefined function available in Pascal that will truncate a real number and store the result in an integer.

Tutorial Quiz 6.0

(1) The ___ statement is used to make a single or compound statement execute a specific number of times.

(2) In successive loop iterations in a ______ loop, the loop control variable is either incremented by one or decremented by one.

(3) The ______ statement is used to select a statement to execute from a list of statements.

(4) The "downto" and "to" are elements of the _______ statement.

(5) An ___ statement must follow the case statement.

DECISION TESTING

Often, it is necessary to make tests to determine the flow of control in a program. The case statement is a simple example. However, it may become necessary to perform more complex tests than the case statement was intended for. Pascal has a powerful set of logical and relational operators that make such testing easy. Most logically complex programs use relational testing for advanced control. The logical and relational operators are as follows :

Logical operators

and	- Will evaluate two boolean expressions, then perform a logical "and" on them, returning either a boolean "true" or "false".
or	- Will evaluate two boolean expressions, then perform a logical "or" on them, returning either a boolean "true" or "false".
not	- Will change a boolean value to the opposite value.

Relational operators

It is often necessary to compare several variables for equality in an expression to determine the flow of control. This may be accomplished by relational testing. There are six relational operators in Pascal, all with equal precedence. Their precedence may be altered just like the arithmetic operators by the use of parentheses. If the relational test fails, a Boolean False is returned by the expression. If the test succeeds, then a true is returned. The operators are as follows.

=	Equal to	<>	Not equal to
>	Greater than	<	Less than
>=	Greater than or equal to	<=	Less than or equal to

(C) copyright 1981 Alcor Systems

There are two constructs in Pascal that often use relational testing for loop control. They are the while and repeat statements. Almost all goto and other branching constructs may be replaced with these statements. Unlike the goto statement, these statements force simple and clear design of loops, often eliminating the unclear conditions for exiting. Usually, if it is not possible to formulate a loop construct using the while, repeat and if statements, instead of a goto, it is because the loop itself has not been properly defined. Ie; the programmer does not have the specifics clear in his mind.

If statement

A typical use of a relational test is illustrated in the if statement. In the following example let the variables "Monday" and "October" be of the type boolean with their values both being true.

Listing 7.1

```
PROGRAM testIF;
VAR
 Monday,October:BOOLEAN;
BEGIN
   Monday:=true;
   October:=true;
   IF October AND Monday THEN
      WRITELN(OUTPUT,'Its October and Monday')
   (*notice no semicolon after the previous statement*)
   ELSE WRITELN(OUTPUT,'Date unknown.');
END.
```

This program will print the message, "Its October and Monday" since October is true, and Monday is true. This example illustrates the use of the " if then else " statement in Pascal. If the expression is evaluated to be true, the first action will be taken. If it is false, the statement following the else will execute. The statements may be simple or compound. Notice that a semicolon may not precede the else in the IF statement.

(C) copyright 1981 Alcor Systems

Notice the following example where "income" has been declared as the type integer and "president" is of type boolean.

```
IF (income > 32000) AND NOT(president) THEN
   BEGIN
   WRITELN(OUTPUT,'You are being audited by the IRS.');
   WRITELN(OUTPUT,'Please justify your deductions.');
   END;
```

The value of the expression will be true if the integer value of "income" is greater than 32000 and the boolean value of "president" is false. When the value of "president" is false, the not operator will reverse its value to true. This type of expression is one of the strengths of Pascal. With a little experience, you will find it easy to write expressions. This greatly improves the readability of logically complex programs. Arguments for relational operators must be of the same type. In the example, "income" must be declared as an integer type for the statement to be valid in Pascal. For now, we will concern ourselves with integer and boolean comparisons.

While statement

The while forces a statement to execute while some condition is satisfied. The condition is the value of a boolean variable or the boolean result of some expression. Some computation inside the loop should change one of the variables used in the test to cause the relational test to fail, terminating the loop. The while statement will perform the test at the beginning of every loop. The while loop might never execute any of the enclosed statements as the initial test occurs before the loop is entered. In the next example, cnt, cost and unitprice are declared as type integer, and underbudget is of type boolean. Notice the following example syntax.

(C) copyright 1981 Alcor Systems

```
cnt:=0;
underbudget:=true;
WHILE (cnt < 20) AND (underbudget) DO
        BEGIN
        cnt:= cnt + 1;
        cost := cnt * unitprice ;
        IF (cost > 200 ) THEN underbudget :=false;
        END;
```

The previous example will execute as a conditional loop instead of a predetermined number of times as in the for loop. When "cnt" gets incremented to twenty one, or cost exceeds 200, the loop will terminate. Note that the "cost > 200" test could have been put in the while expression just as easily.

Repeat statement

Another statement similar to the while is the repeat. A statement or series of statements will be repeated until an expression becomes true. The difference between the while and repeat may not be obvious. The difference is that the repeat statement will always execute at least once because the relational test occurs at the end of the loop. The use of repeat sometimes causes problems for new programmers, as there may be cases where you do not want the loop to execute at all, however it will always execute at least once. An example of repeat is as follows:

```
cnt:=0;
underbudget:=true;
REPEAT
   cnt:=cnt + 1;
   cost := cnt * unitprice;
   inventory:=inventory +1;
   IF (cost > 200) then underbudget:=false;
UNTIL(cnt >= 20) OR NOT(underbudget);
```

Notice that the test was changed to use the OR operator instead of the and operator. This is simply due to the different context of the two statements. There is no begin or end required. The statement(s) to be executed are simply placed between the repeat and until. What happens to this loop if the initial value of "unitprice" is greater than 200 ? The loop will terminate on the first iteration, but alters the value of "inventory". This might not be the desired result and could cause an illegal entry into the inventory. In this situation, the while statement would be the proper choice of a looping construct, as it would detect this before "inventory" is changed.

Tutorial Quiz 7.0

(1) The logical operators in Pascal are: and, ___ , ___ .

(2) The and operator will return a value of ____ , if the value of the both expressions it is evaluating is true.

(3) The or operator will return a value of _____, if one of the expressions it is evaluating is true.

(4) The not operator will reverse the value of a _______ variable or expression.

(5) The IF statement will execute the else portion of the statement, if the value of the expression is ______.

(6) The while statement will execute as long as the boolean result of the expression is _____.

(7) The repeat will execute all statements between the repeat and until as long as the expression is _____ .

(C) copyright 1981 Alcor Systems

Chapter 8

PROCEDURES AND FUNCTIONS

Procedures

In the introduction, one of the claimed strengths of Pascal was that it promotes modularity. Modularity is another name for organizing a program into sections, each of which performs a specific function, instead of one large block of continuous statements. One of the reasons that Pascal programs may have a high degree of modularity is that the language was designed with procedures and functions in mind. In a few languages, they are not even supported, and in others, passing parameters can become a major chore. This is not the case in Pascal, as several different methods are available to pass data to subroutines that need it. Furthermore, there are rules about how procedures may call other procedures and access their internally defined variables. These scoping rules, as they are called, may seem a little restrictive, but they provide valuable protection. This partitioning of the problem eventually decreases program size and improves readabilty to the programmer or anyone who must maintain it. A simple way to decide whether a procedure or function should be used is to examine the problem and to decide if there are a series of statements that need to be executed several times, and in different parts of the program. The identified program segments should be placed in a procedure or function.

Procedure structure

Procedures may be thought of as complete sub-programs that have data passed to them as needed. In many descriptions written about Pascal, they are often called one of the basic blocks, and in this manual, a block will be considered to be a program, procedure or function. The structure of a procedure is the same as for the original program with a few exceptions. The data that is passed to a procedure block is passed through a parameter list. The parameter list is placed after the procedure name. Examine the following program.

```
Listing 8.1
PROGRAM INSTRUCTIONAL;
VAR
   number   :INTEGER;
   posnumber:INTEGER;
   legal    :BOOLEAN;
PROCEDURE readn ( VAR number: INTEGER; VAR legal: BOOLEAN );
(* The purpose of this routine is to read        *)
(* a positive number from a file in a            *)
(* character format and convert it to an integer*)
(* format.                                       *)
VAR
    loopcontrol,forcntr,inc:INTEGER;
    string                 :ARRAY [1..72]OF CHAR;
BEGIN
    FOR loopcontrol :=1 to 72 DO string[loopcontrol]:=' ';
    loopcontrol :=0;
    WHILE NOT EOLN(INPUT) DO
       BEGIN
       loopcontrol := loopcontrol + 1;
       READ(string[loopcontrol]);
       IF(string[loopcontrol]=' ')THEN
         (* Remove all leading blanks from array *)
         loopcontrol:=loopcontrol - 1;
       END;
    number:=0;
    inc:=1;
    FOR forcntr :=loopcontrol  DOWNTO 1 DO
       BEGIN
       number:=number+((ord(string[forcntr])-ord('0'))*inc);
       inc:=inc*10
       END;
    IF (number < 0) THEN
       BEGIN
       legal := false;
       WRITELN('* Error - Illegal entry. Try again. ');
       END
   ELSE legal:= true;
END;                (*procedure readn*)

BEGIN
legal:=false;
WHILE NOT legal DO
   BEGIN
     WRITELN('Enter any positive number:');
   READN(posnumber,legal);
   END;
END.
```

The purpose of the program 8.1 is to read a positive integer in from the file input and to check for illegal entries. This declared procedure represents a typical use for a procedure, since it might be called several times, from different places in the program. Notice the eoln(input) . Eoln is a boolean function that will return a true value when an "end of a line" of the file specified in the parentheses is reached. As soon as the cursor is moved from this position by another readln, it's value becomes false again. Notice also the function call to ORD. ORD is a Pascal function that returns the internal integer representation of a character.

Since there is only one copy of this procedure in memory no matter how many calls there are, a considerable amount of memory space can be saved. In fact, a procedure's variables do not occupy storage space until the procedure is actually called.

The procedure declaration comes after the const and var section, and before the first begin statement of the block in which it resides. Remember, a block may be a main program, procedure or function.

Local variables

Local variables are variables declared in a particular procedure, function, or program. For example, the variable "forcntr" declared in procedure readn, is local to "readn" and is accessible from "readn" only. However, notice the variable "number" declared in the main program block. Inside the procedure "readn" , the variable "number" may be used without declaring it, since it appears in the calling program. This means that if "readn" is called by the main program and "readn" alters "number", then upon return to the main program, "number" will have the altered value. This side effect can be avoided by declaring "number" again in the procedure block. Then all references to "number" will refer to a different variable. The use of global variables should always be kept to a minimum, so as to minimize any accidental changes in their values.

Procedure parameters

An alternate method of changing global variables within a procedure is to pass them as parameters in a parameter list. This allows different variables to be passed at different times and makes the use of the global variable more visible in the program. The parameter list is placed in the procedure declaration after the keyword procedure. In the parameter list, a variable may be passed by two different methods. These two methods are referred to as passing by reference, or passing by value.

When a parameter is passed by reference, the actual argument is passed to the procedure, and if the procedure alters its value, the argument's value is changed in the rest of the program. When a parameter is passed by reference, the argument must be a variable.
When a parameter is passed by value, what is passed is a copy of the argument. If the procedure alters the parameter's value, the value in the rest of the program is not changed. When a parameter is passed by value, the argument may be a variable or any legal arithmetic expression. Parameters passed by value will prevent unwanted changes in a variable value by the called procedure. Notice the following example parameter list.

```
PROCEDURE test ( date: INTEGER; VAR profit:REAL; cost:REAL);
```

The variables "cost" and "date" will be passed by value. The variable "profit" will be passed by reference. Every time a variable is to be passed by reference, the keyword "var" must precede it, otherwise it will automatically be passed by value.
It is sometimes hard for new programmers to understand the difference between letting variables be global when accessing them in a procedure, versus passing them by reference. There is a major difference, in that different variables may be passed to a procedure. The only stipulation is that the variables must match the parameter list. If they are declared as globals and alterd by a procedure, then all values to be passed to the procedure must be transferred to these global variables. A second major difference is that in large programs, it is often difficult to determine what routines are changing specific variables. Sometimes accidental changes may occur in global variables. These changes are often referred to as side effects.
By adhering to the convention of passing the variables to a procedure, it is easier to determine how procedures alter external variables and to minimize unwanted side effects. Certainly, global variables do have use in Pascal programs, but many new Pascal programmers have a tendency to over use them.

Calling procedures

Procedures are called simply by referencing their name followed by an argument list enclosed in parentheses. The list should be composed of variables of the same type and order as declared in the procedure declaration section.

(C) copyright 1981 Alcor Systems

Functions

Another block in Pascal similar to the procedure is the function. Its internal structure is the same as the procedure with const, var and type sections optional. The purpose of a function is similar to a procedure. A procedure may stand alone as a statement, as the call to "readn" illustrates in program 8.1 . A function may not stand alone. It must be used in an expression, and may be used anywhere a variable can be used. Consider the following program.

Listing 8.4

```
PROGRAM functiontest;
VAR
   num:INTEGER;

FUNCTION ABS( number: INTEGER ) : INTEGER;
BEGIN
IF (number < 0) THEN ABS := - number
   ELSE ABS := number;
END;

BEGIN
   num:= -30;
   num := ABS (num);
   WRITELN(' the absolute value of num = ',num);
END.
```

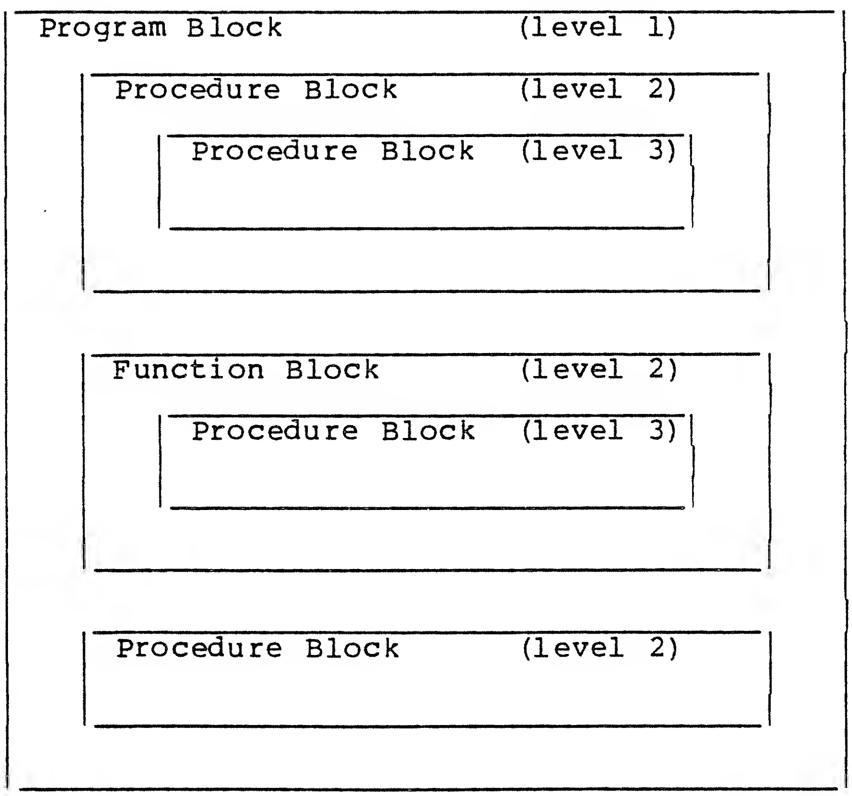
The result returned by the abs function is of the type integer. The result must be used in an expression or assignment statement. It is not valid to simply say abs (num) . The mechanism used to transfer the functions calculated value back to the calling program is the use of an assignment statement to assign the value to an identifier that has the same name as the function name. This particular function is already predefined in Pascal, and serves the same purpose as the example.

(C) copyright 1981 Alcor Systems

Advanced program structure

Pascal is a block structured language. This means that a program is constructed in a block like manner. At a minimum, a program consists of one block. More blocks are created through the use of procedures and/or functions by placing them inside this outermost "program block". The term for this process is called nesting. The rule for nesting is that a block may lie entirely within another block, but blocks do not overlap in any other way. A level of nesting can be assigned to each block of a program. This provides an appropriate tool for describing scope rules which are discussed later. The block structured organization of a program can be represented pictorially by the following diagram.

Diagram 8.1



A program then consists of at least one block, the program block, and optionally it contains procedure and/or function blocks which are nested within.

(C) copyright 1981 Alcor Systems

Local variables are those variables declared within the var section of a particular procedure. Locals can be accessed from the body of the procedure in which they are declared and from those procedures declared within it. If a variable is used within a procedure and is not declared local to it, then a global variable is used. Global variables are those variables declared in an outer enclosing block.

Listing 8.2

```
PROGRAM globals;
VAR
   i : INTEGER;
   b : BOOLEAN;

PROCEDURE inner;
VAR
  b : INTEGER;
BEGIN
   b := i + 25;
   i := i + 1;
END;

BEGIN
   i := 0;
   writeln(i);
END.
```

In the above program, the i in the procedure refers to the variable i in the main program. Since the program "global" is an enclosing block to the procedure "inner", the variables declared within the program are accessible to the procedure. In the case of the variable "b", the var section of the procedure redeclares b to be an integer. When b is referred to in the procedure inner, the local variable is used. The declaration of b as a local variable "masks" the global definition of b.

Scope rules

The rules of accessability of variables, types and constants are referred to as scope. The scope of an identifier is the procedure in which it is declared, and all procedures declared within that procedure. All identifiers including types, constants, variables and procedure declarations have scope.

If an identifier is redeclared within its scope, the outer definition becomes inaccessible within the scope of the inner definition. In the example above, the declaration of b as an integer within the inner procedure causes all references to b in that procedure to refer to the local variable. The outer definition of b as a boolean cannot be seen.

Pascal requires that all identifiers be declared before they are used. If the declaration of an identifier has not yet been encountered in the text of a program, then the identifier is considered undefined. A procedure can be called from the body of the block declaring it, from the procedures declared within it and from the procedures declared within the same block. However, if procedure A is declared before procedure B in the same block, then procedure B can call A, but procedure A cannot call B. This is due to the fact that the declaration of B has not been encountered in the source text when the body of procedure A is being compiled.

The above visibility restriction can be avoided with the use of forward declarations. In a forward declaration, the body of the procedure is replaced with the word FORWARD. The actual body is then supplied later. If all procedures within a block are declared forward, then any one of them can call any other.

Listing 8.3

```
PROGRAM Outer;
VAR
  i : INTEGER;
FUNCTION Distance(xl, x2 : INTEGER):INTEGER;FORWARD;
FUNCTION Abs(tvalue : INTEGER) : INTEGER; FORWARD;

FUNCTION Distance(*(xl, x2 : INTEGER) : INTEGER *);
BEGIN
   distance := abs(x2 - xl);
END;

FUNCTION Abs(*(tvalue : INTEGER) : INTEGER *);
BEGIN
   if tvalue < 0 then abs := -tvalue
      else abs := tvalue;
END;

BEGIN
   WRITE('DISTANCE = ',Distance(8,2));
END.
```

If a procedure is declared forward, its parameter list is supplied by the forward declaration. The body appears later in the text. The body is introduced by the procedure name followed by a semicolon. The parameter list is not repeated. Notice that in the example, the parameter list is commented out by putting (* *) around it. It is good practice to include the parameter list of a forward procedure in a comment. This makes the body of the procedure easier to read.

Tutorial Quiz 8.0

(1) __________ and _________ promote modularity and functionality in programs.

(2) Data is passed to procedures and functions through a ________ list.

(3) Blocks may be ______ within other blocks.

(4) Nesting affects the _______ of blocks.

(5) A block nested within an outer block may access the outer blocks _________ .

(6) Parameters may be passed by value or _________ .

(7) When a variable is passed by ______ , a copy of the variable is passed.

(8) When a variable is passed by reference the keyword ___ must precede it in the parameter list.

(C) copyright 1981 Alcor Systems

ADVANCED DATA TYPES

Array data type

Another cousin to the data types already explained is the array. Sometimes a large number of variables of a particular type are needed. If for example you required seventy two variables of the type char to represent a user's input character string from a file, you could declare them as previously explained. The disadvantage is obvious, as the effort would be time consuming. Furthermore, accessing the individual variables would be confusing, as each would have a different name.

There is a simple answer to this problem and it is the data type array. You may declare a variable as :

```
variablename : array [1..n] of type;
```

where type is any previously defined data type and n is the number of variables desired. For now we will concern ourselves with integer dimensions. Integer dimensions may be any positive or negative numbers such that the range of dimensions do not cause a storage overflow. This is a machine dependent constraint that varies among implementations. Thus we may declare :

```
VAR     line : array  [1..72] of char;
        varname:char;
```

To access a component of this array you would use a subscript denoting the numerical element. An example assignment might be varname := line[4]; . Varname would be set to the value of the fourth element in the array line.

Any array may be declared with the word PACKED as a prefix. The packed attribute tells the compiler to store the data elements as efficiently as possible. In Standard Pascal, you may not pass elements of packed structures by reference to procedures or functions, and packed elements may not be used as arguments in READ statements. In Alcor Pascal, there are no such restrictions.

(C) copyright 1981 Alcor Systems

Arrays

```
listing 9.1

PROGRAM onedimarray;
VAR
   stringl: PACKED ARRAY [1..72] OF CHAR;
BEGIN
   WRITELN('Enter command string');
   READLN(INPUT,stringl);
   WRITELN(OUTPUT,stringl);
   WRITELN('Program complete');
END.
```

If the I/O is directed to the terminal, the program will display the prompt: "Enter command string" . At this point the user may type up to seventy two characters of input, terminated with the return key. The input characters will be input to the array "stringl" left justified. If the input character string is less than seventy two characters in length, the remaining storage positions in the array will contain blanks. At this point the input message will be echoed to the terminal. In Alcor Pascal, an entire single dimension packed or nonpacked array of char may be input/output by a single read/write.

Arrays in Pascal may have multiple dimensions. Suppose that you had a number of input character strings as in the previous example, and it was desired to store every character string. A simple answer would be to declare the array line : array [1..5,1..72] of char; . This declaration is a Pascal short form for the declaration of array[1..5] of array[1..72]of char; .

If the data structure is a two dimensional array of char, then the read command will not input the entire array automatically, but instead requires that each individual sub-array be read in with a separate read statement.

(C) copyright 1981 Alcor Systems

Remembering that any single dimension array may be input by a single READ statement, leads to the following example array input sequence. Examine the following program.

listing 9.2

```
PROGRAM arrayIO;
VAR
   I          : INTEGER;
   string1    : ARRAY [1..5,1..72] OF CHAR;
BEGIN
   FOR I:= 1 TO 5 DO
       BEGIN
       WRITELN(OUTPUT,'Enter command line  ',I);
       READLN(INPUT,string1[I] );
       END;
END.
```

This program will prompt the user for five different command lines. In each case, the individual sub arrays are loaded into the array by the program.

Since individual array elements are of the type char, any operations that can be performed on a simple variable, of type char, may be performed on an array element. Remember also that arrays may be of any type such as boolean, integer or any user defined data types, including arrays. Arrays may have upper and lower bounds declared as constants in the declaration, and in fact, the name of most simple data types may be substituted for the bounds. The number of array elements in this case is determined by the number of elements in the data type.

(C) copyright 1981 Alcor Systems

User defined data types

The data types explained so far have been pre-defined. In Pascal, you may define new data types at will. These defined types have names chosen by the programmer and are declared in the TYPE section. Once declared, they may be used where predefined type names are allowed. This is a very powerful feature. Take for example the case where a programmer is manipulating an integer variable in Basic that may take on one of four values, 1..4. The numbers may represent the colors red, green, blue and orange. When the value is 1 : a message is written to the terminal saying that the color red is being processed, 2 : That the color green is being processed and so on. This is typically known as decoding information from a variable's value. Needless to say, when Basic programs get very long, it is difficult to determine their flow because of this decoding and encoding of information. A simpler way would be to declare a variable that could take on the value of red, green, blue and orange. Then tests could be performed to see if the value of the variable is red, etc. Program logic would be much clearer and easier to follow. In fact, this is exactly what the following program does.

listing 9.3

```
PROGRAM usertypes;
TYPE  difcolor = (red, green, blue, orange);
VAR
   color: difcolor;
BEGIN
   color := red;
   REPEAT
      CASE color of
         red    : WRITELN(OUTPUT,' The color is red');
         green  : WRITELN(OUTPUT,' The color is green');
         blue   : WRITELN(OUTPUT,' The color is blue');
         orange : WRITELN(OUTPUT,' The color is orange');
         END;
   color := succ(color);
   UNTIL( color = orange );
END.
```

(C) copyright 1981 Alcor Systems

Enumerated user defined types

Program 9.3 illustrates an enumerated user defined type, "difcolor" . An enumerated type is where a list of possible variable values are given in the type declaration. The predefined function, "succ" is available in Pascal, and is a convenient way of incrementing a user defined variable type to the next possible value. In a simple program using an integer variable, this could be accomplished by adding one to the variable, but this would not make sense with a user defined type. User defined enumerated data types may not have their values written out. Program 9.3 gives an example of how that may be accomplished.

Subrange types

A variable may assume a value that is in a sub- interval of some other simple type. In this case, it may be declared to be a subrange type. For example, integer may represent all whole numbers between -32,768 and 32,767. In the type section, a subrange user defined type might be declared to be byte = 0..255 ; I.E.; any variable of the type byte may take on the value from 0 to 255 . The same operations may be performed on a subrange type that are applicable to the original type. Also a subrange type may be the subrange of any user defined simple type.

listing 9.4

```
Program subrange;
TYPE
   baddate            = 1900..1903;
   uppercaseletters = 'A'..'Z';
   lowercaseletters = 'a'..'z';
   digits             = '0'..'9';
   xaxis              = -100..100;
VAR
   testyear           : baddate;
   upperletter        : uppercaseletters;
   lowerletter        : lowercaseletters;
   digit              : digits;
BEGIN
END.
```

All of the above examples are valid subrange declarations. Named subrange types are very helpful when a programmer wants to clearly identify the data differences between specific variables to increase readability. Also, the storage required for a subrange variable is proportional to the interval it spans. This may be important when building large data structures to be implemented on microcomputers.

(C) copyright 1981 Alcor Systems

RECORD data types

So far, the only structured data type examined has been the array. The array is an excellent mechanism for storing large amounts of data of the same TYPE. For example, the series of text strings input from the terminal were efficiently stored using arrays of CHAR, and any individual character was easily accessible. However, it is often desired to keep variables of different data types grouped together. Take for example, a list of a business's customers along with vital information about each customer. Suppose that you desired to keep the following information about every customer:

```
Name
Customer category
Mailing address
Telephone number
Dollars spent in store
On catalog circulation list
```

This might represent a situation where the business would like to keep a data base updated. In languages like Basic, the only way to maintain this information would be multiple arrays containing encoded information. This is not the case in Pascal, as you may build a RECORD which can store all of the above information in a clear and concise format. Furthermore, you may declare an array to be of this user defined type.

Record data types

In Pascal, a RECORD is a predefined data structure which is composed of component variables. These component fields may be variables of any Pascal predefined, or user defined data types. The purpose of a record is to group variable information into logical entities, such that any particular component field may be operated on, or the entire record may be referenced as a whole. The following is an example of how the previous business record is declared in Pascal.

(C) copyright 1981 Alcor Systems

```
Listing 9.5

PROGRAM database;
TYPE
  custmrcategory = (business,individual);
custmrecord = RECORD
  custmrtype   : custmrcategory;
  address        : PACKED ARRAY[1..72] OF CHAR;
  telephone      : PACKED ARRAY[1..15] OF CHAR;
  expenditures  : REAL;
  cataloglist    : BOOLEAN;
  END;
VAR
   custmr     : custmrecord;
   custmrlist : ARRAY[1..100] OF custmrecord;
   index        : INTEGER;
   ans          : CHAR;
PROCEDURE custmrinp( VAR custmr:custmrecord);
VAR custyp : CHAR;
BEGIN
  WRITELN('* Enter customer type: (business/individual)');
  READLN(custyp);
  IF(custyp='I')THEN
    custmr.custmrtype:=individual
  ELSE custmr.custmrtype:=business;
  WRITELN('* Enter address:');
  READLN(custmr.address);
  WRITELN('* Enter telephone number:');
  READLN(custmr.telephone);
  WRITELN('* Enter expenditure in dollars:');
  READLN(custmr.expenditures);
  WRITELN('* Want on catalog circulation list: (true/false)');
  READLN(custmr.cataloglist);
END;
BEGIN
  index:=0;
  ans:='N';
  WRITELN('** BUSINESS XYZ CUSTOMER RECORD PROGRAM **');
  WHILE (ans <> 'S') DO
    BEGIN
    index:=index+1;
    custmrinp(custmrlist[index]);
    WRITELN('* MORE CUSTOMERS (STOP/CONTINUE)');
    READLN(ans);
    END;
END.
```

(C) copyright 1981 Alcor Systems

The outer shell that must enclose record type declarations is of the form:

```
type name = RECORD
            END;
```

The component field declarations reside between the RECORD and END; . The field declarations are defined in the same way as the VAR section of the program. In program 9.5, the user defined record name is custmrecord. The component field declarations: custmrtype, address, telephone, expenditures, and cataloglist are defined exactly the same way as the the program variables are in the VAR section. All the field components belong to the data type custmrecord. Since custmrecord is treated like any other user defined type, we may now declare a variable to be of type custmrecord in the program VAR section.

The difference between a record and other simple user defined data types is that there are component fields in a record that are really variables themselves. In example 9.5, the variable custmr is of a record type. When referring to custmr in expressions, to reference the entire record, you simply use the variable name, custmr. To access the component field, expenditures, you would prefix expenditures with the record variable name, custmr, separated with a '.' character. Example:

```
custmr.expenditures:= 99.95;
```

If another record named excustomer had been declared, the following would be a valid statement.

```
excustomer:=custmr;
```

In this case, all component fields in excustmr would be set to the component fields in custmr. Variables of type record, and their associated component fields, obey the same rules for use as all other typed variables.

The purpose of program 9.5 is to perform record I/O utilizing the predeclared text files input and output. Notice the read and write statements utilize record component fields as arguments. Read and write behave as though the component fields were variables declared in the VAR section. As with other variables, I/O may not be performed to a text file through a component field that is of a user defined enumerated type.

WITH statements

The use of records may often cause segments of the program that reference them to become long and tedious, because every time a component field is referenced, the record variable name must precede it. Accessing component fields may be simplified by using the WITH statement. Examine the following procedure, which could be included in program 9.5 .

Listing 9.6

```
PROCEDURE custmroutput(VAR custmr:custmrrecord);

BEGIN
   WRITELN('** CUSTOMER OUTPUT RECORD FOR BUSINESS XYY **');
   WITH custmr DO
        BEGIN
        IF(custmrtype=business)then
           WRITELN  ('Customer type    : Business')
        ELSE WRITELN('Customer type    : Individual');
        WRITELN     ('Address          : ',address);
        WRITELN     ('Telephone        : ',telephone);
        WRITELN     ('Expenditures     : ',expenditures);
        WRITE       ('Circulation list : ');
        IF (cataloglist)THEN WRITELN('Yes')
        ELSE WRITELN('No');
        END;
END;
```

The action of the WITH statement is to eliminate the normally required record variable name prefix when accessing component fields of that record. The scope of the WITH is one statement, which in this case is a compound statement.

(C) copyright 1981 Alcor Systems

File of TYPE

INPUT and OUTPUT are examples of TEXT files in Pascal. These FILE types have been used for all of the program examples so far. A TEXT file is Alcor Pascal predeclared to be a special file of char, with rules for performing I/O using INTEGER, REAL and BOOLEAN variables. In Alcor Pascal, there are extensions to allow for performing I/O using ARRAY variables in text files.

A FILE OF <any known type> may be declared in Pascal. Files of types other than text are primarily used for storing data which will be retrieved at some other time. For example, a FILE OF customerecord could be defined in the type section. (customerecord as defined in listing 9.5) A variable of type customerecord could be written to this file. The important thing to remember is that an entire record may be written (or read), by one I/O statement. Component fields of this record may not be read or written individually to a file of records. When I/O is performed with a FILE OF <any type except text>, no ASCII encoding or decoding of information takes place. Instead, the binary representation is used. This is not particularly useful when the I/O is directed to a terminal, but is effective for storing large amounts of information on disk media. The predeclared procedures WRITELN and READLN are not valid when performing I/O with a file of any type except TEXT, although read and write perform normally. The program in the appendix of this manual utilizes a FILE OF customerec for storing information in a data base. This is a typical use for a FILE OF TYPE.

(C) copyright 1981 Alcor Systems

(1) If a large number of variables of the same TYPE need to be declared, the ______ may be the correct data structure to use.

(2) Arrays in Pascal may have more than _____ dimension.

(3) New user defined _____ may be declared in Pascal programs.

(4) An __________ TYPE is defined by a list of identifiers given to be the different values allowable for a variable.

(5) A _________ TYPE is any user defined TYPE that is a sub-interval of another simple TYPE.

(6) A _________ TYPE is used to logically group together data of different types.

(C) copyright 1981 Alcor Systems

DYNAMIC DATA TYPE

All of the variable types discussed so far have been "static" in nature. This means that the size of data structures such as the array have to be defined before the program is compiled or executed. In program 9.5, the size of the array customer list has an upper bound of 100 entries. If more than 100 storage locations are needed to store the customer records, the array declaration has to be changed in the source program, and the source recompiled. In most popular micro-minicomputer Pascal implementations today, there are limits to the number of storage locations that may be declared in a program. This limitation is usually proportional to the size of the program in conjunction with the type and number of variable declarations. It is usually impractical due to these memory restrictions to declare arrays and other data structures to be larger than required. The static nature of variable declarations often create problems in some programming applications. Suppose for example, that in program 9.5 it was desired to keep a list of sales transactions for each customer attached to each customer record. This could be accomplished by declaring a component field of each customer record as being an array of transaction records. Then at any time you could access the sales transaction of every customer. This would require that the number of sales transactions per customer be limited to a preset number by the array declaration. It might be feasible to limit the number of customers to 100, but the number of transactions per customer might vary. There is a mechanism in Pascal to allow for dynamic variable allocation at program execution time. It is possible to request a new storage location for a variable by calling the Pascal pre-defined procedure NEW.

Procedure NEW

By calling the procedure NEW, it is possible to get a pointer to a memory storage location that is the proper size for the argument variable. It is important to remember that the same limitations on the amount of memory available still apply, however dynamic allocation of memory allows for better utilization of space. The variable used as an argument for the NEW procedure call must have been declared in the VAR section. It must be declared as a TYPE that is a pointer to the actual data type. An example of a pointer data type declared in the TYPE section is as follows:

(C) copyright 1981 Alcor Systems

```
 Listing 10.1

TYPE
     trxptr = ^trxrec;

trxrec = RECORD
   nexttrx: trxptr;
   invoicenumber   : INTEGER;
   date            : ARRAY [1..10] OF CHAR;
   transprice      : REAL;
   partnumberlist  : ARRAY [1..10] OF CHAR;
   END;

VAR
   trx : trxptr;
```

In the example program segment, the variable trx is of type trxptr. In the type section, trxptr is defined to be a pointer to " trxrec " . The character " ^ " denotes a pointer in Pascal. Therefore, the variable trx is a pointer to a storage location in memory of the size required to store the RECORD trxrec. This storage location may be requested anytime during program execution as opposed to program startup. Pointer types to large data structures may be declared in a program with minimum memory space penalty until the procedure NEW is called during program execution. Notice at the "^trxrec" point in the type declaration, trxrec has not been defined. In Pascal, declaring a pointer to an as yet undefined type is valid.

The following program segment illustrates a few simple methods of using pointer variable types.

```
Listing 10.2

PROGRAM dynamic;

(* TYPE declaration section from listing 10.1 *)

VAR
  trx      : trxptr;
   nexttrx : trxptr;
BEGIN
   NEW(trx);
   trx^.invoicenumber:=2345;
   trx^.transprice   :=99.95;
   nexttrx:=trx;
   WRITELN('* Transaction invoicenumber : ',
          trx^.invoicenumber);
   WRITELN('* Transaction price         : ',
          trx.transprice);
   DISPOSE(trx);
END.
```

If the pointer itself is being referenced, just the variable name is used. In the example, the pointer variable nexttrx is set to the value of trx. When referring to the contents of the storage location, an " ^ " follows the variable name. "trx^.transprice" refers to the value of the component field stored at that location. These basics of pointer data type manipulation are used to build "linked lists" . A linked list is a chained list of dynamic storage areas.

Notice the procedure call to DISPOSE. The purpose of DISPOSE is to release the storage area acquired in the NEW call. After the DISPOSE, the data stored at the dynamic memory location is effectively lost. This is an important feature of Pascal. Careful use of NEW and DISPOSE can result in programs that dynamically grow and contract in memory size as needed, and efficiently manage the computer resources.

LINKED LIST

A linked list is a programming technique that chains together a series of variables. A thorough discussion of linked list processing would entail several chapters, and is really a topic for a data structures book. It will be covered briefly here because it is integral to discussions about dynamic memory management.

(C) copyright 1981 Alcor Systems

In example 10.1 , the data type trxrec has a component field which is a pointer to a storage area of the same type as itself. A pointer to another record node may be stored in this field. In the record pointed to, a pointer to another record node could be stored and so on.. . In this way, a series of record nodes may be linked together. The following diagram will help to visualize this list.

Listing 10.3

```
VAR
   headnodeptr : trxptr;

                          headnodeptr
                               |
 ______________________________V________________________
|                    record number 1                    |
|   trxrec = RECORD                                     |
|      nexttrx: trxptr;                            --------------
|      invoicenumber   : INTEGER;                       |      |
|      date            : ARRAY [1..10] OF CHAR;         |      |
|      transprice      : REAL;                          |      |
|      partnumberlist  : ARRAY [1..10] OF INTEGER;      |      |
|      END;                                             |      |
|_______________________________________________________|      |
                                                               |
 _______________________________________________________       |
|                    record number 2                    |<-----
|   trxrec = RECORD                                     |
|      nexttrx: trxptr;                            --------------
|      invoicenumber   : INTEGER;                       |      |
|      date            : ARRAY [1..10] OF CHAR;         |      |
|      transprice      : REAL;                          |      |
|      partnumberlist  : ARRAY [1..10] OF INTEGER;      |      |
|      END;                                             |      |
|_______________________________________________________:      |
                                                               |
 _______________________________________________________       |
|                    record number 3                    |<-----
|   trxrec = RECORD                                     |
|      nexttrx: trxptr;                            --------------
|      invoicenumber   : INTEGER;                       |      |
|      date            : ARRAY [1..10] OF CHAR;         |      |
|      transprice      : REAL;                          |      |
|      partnumberlist  : ARRAY [1..10] OF INTEGER;      |      |
|      END;                                             |      |
|_______________________________________________________:      |
                                                               |
                         NIL <----------------------------------
```

The variable headnode is a pointer variable declared in the VAR section of the program. At some point in the program, a NEW procedure call could be made with headnode as its argument. Headnode would now be a pointer to the start of the list. Notice the word NIL at the end of the list. NIL is a reserved word in Pascal. This simply sets the pointer to an initialized value that may be tested for in looping statements. A word of caution when using pointers in Pascal. If a pointer variable has been declared, but not set to any value, there is no guarantee of its value. It will not necessarily be set to NIL . Most Pascal implementations do not perform a runtime check for uninitialized values. Use of uninitialized pointers can lead to the program writing over itself in memory with execution becoming unpredictable. These kinds of programming errors will not show up at compile time, and can be extremely hard to find during program execution. The following segment illustrates how list 10.3 could be built.

Listing 10.4

```
PROGRAM linkedlist(input,output);
TYPE
   trxptr = ^trxrec;
   textline       =  PACKED ARRAY [1..10] OF CHAR;
   trxrec = RECORD
      nexttrx: trxptr;
      invoicenumber   : INTEGER;
      date            : textline;
      transprice      : REAL;
      partnumberlist  : textline;
      END;
VAR
  headnode,transnode:trxptr;
                  I : INTEGER;
PROCEDURE readtrx(VAR trx:trxrec);
  (* The purpose of this routine is to prompt the user for *)
  (* the purchaser's trx record                            *)
BEGIN
  WITH trx DO
      BEGIN
      WRITELN('ENTER INVOICE NUMBER:');
      READLN(invoicenumber);
      WRITELN('ENTER DATE:');
      READLN(date);
      WRITELN('ENTER TOTAL PURCHASE PRICE:');
      READLN(transprice);
      WRITELN('ENTER PARTNUMBER(S) SEPARATED BY COMMAS:');
      READLN(partnumberlist);
      END;
END;   (*readtrx*)
```

(C) copyright 1981 Alcor Systems

Listing 10.5 (continuation 10.4)

```
PROCEDURE writetrx(VAR trx:trxrec);
(* The purpose of this routine is to write the purchaser *)
(* trx entry                                             *)
VAR   I:INTEGER;
BEGIN
   WITH trx DO
      BEGIN
      FOR I:= 1 TO 35 DO WRITE('*');
      WRITELN;
      WRITELN('INVOICE NUMBER       : ',invoicenumber);
      WRITELN('DATE                 : ',date);
      WRITELN('TOTAL PURCHASE PRICE : ',transprice:10);
      WRITELN('PART NUMBER LIST     : ',partnumberlist);
      FOR I:=1 TO 35 DO WRITE('*');
      WRITELN;
      WRITELN;
      END;
END;  (*PROCEDURE writetrx*)

PROCEDURE listrxs( temptr : trxptr );
(* the purpose of this procedure is to traverse the linked*)
(* list attached to the argument pointer, writing the     *)
(* values of the trx records                             *)
VAR
  loctrx : trxrec;
BEGIN
  (* traverse trx linked list,writing trxs *)
WHILE (temptr <> NIL) DO
  BEGIN
    (* load the contents of localtrx with the *)
    (* contents of temptr                       *)
  loctrx:=temptr^;
  writetrx(loctrx);
    (* set temptr to the next node in the linked list  *)
  temptr := temptr^.nexttrx
  END;
END;   (*listransactions*)
```

Listing 10.6 (continuation 10.5)

```
BEGIN     (* begin main program linkedlist *)
  (* initialize pointer that will always reflect the *)
  (* beginning of the list.                                  *)
  (* this will set the end of the list to NIL during the first  *)
  (* pass through the FOR loop                               *)
headnode := NIL;
  (* read 3 trxs and link each new one to the beginning *)
  (* of the list                                            *)
FOR I := 1 to 3 DO
  BEGIN
  NEW(transnode);
    (* insert the newnode in front of the old headnode *)
    (* link to the old headnode                         *)
  transnode^.nexttrx := headnode;
    (* make the newnode the new headnode                *)
  headnode := transnode;
    (* load the actual data into the fields of the new node *)
  readtrx(transnode^);
  END;
  (* list all trxs entered *)
listrxs(headnode);
END.  (*main program*)
```

(C) copyright 1981 Alcor Systems

Sets

Sets in Pascal have the same meaning as they do in the normal mathematical sense. If a group of objects are declared in set A, and a group of objects are declared in set B, a number of operations may be performed on these sets such as :

(1) Membership and relational testing
(2) Set arithmetic (union,intersection,difference)

In the case of Pascal, the objects are simply data values. These data values may be Pascal predefined or user defined. An example would be a SET OF CHAR, or a SET OF digits where digits is a user defined subrange type of CHAR. Testing could be performed to see if the SET OF digits is in the SET OF CHAR if desired. The method of declaring set variables is :

```
VAR    A,B : SET OF <type> ;
```

This means that A and B may contain from one to all of the data values declared by the type, however its membership is undefined until it is initialized like any other variable. In the body of the program, a set may be initialized to empty by :

```
A := [];
```

Membership testing

Once the set variables are initialized, a series of BOOLEAN relational tests may be performed. The relational operators are as follows:

set1 = set2	Set equality- If (all members of first set are in the second set and all members of second set are in the first set) : returns true.
set1 <= set2	Subset- If (all members of first set are in the second set) : returns true.
set1 >= set2	Superset- If (all members of second set are in the first set) :returns true.

(C) copyright 1981 Alcor Systems

set1 <> set2 — Set inequality- If all members of first set are in second set, and all members of second set are in first set : returns false.

Individual element membership may be tested by using the IN operator. If a variable had been declared of the same type as the base set type, the IN operator may be used to check for set membership. An example would be:

```
Listing 12.1

TYPE
  DIGITS = '0'..'9';
VAR
        DIGIT  : SET OF DIGITS;
        D      : CHAR;

 BEGIN
   D:='a';
   DIGIT:=['0'..'9'];
   IF(D IN DIGIT)THEN DO (*action*);
   IF(D='0')OR(D='1')OR(D='2')OR(D='3')OR(D='4')OR(D='5')
     OR(D='6')OR(D='7')OR(D='8')OR(D='9') THEN DO (*ACTION*)
```

The two IF statements in the above program segment are equivalent. Notice that the equivalent IF statement using sets is a more concise and readable statement. This represents a simple use for sets for the average programmer.

Set arithmetic

There are three set operators in Pascal. Each requires two arguments. Arguments should be sets of the same base type, and the result will be of the same type. The operators are:

A + B	Gives the union of A and B
A * B	Gives the intersection of A and B
A - B	Gives the difference of A and B.

(C) copyright 1981 Alcor Systems

The following segment program illustrates set operator use.

```
LISTING 12.2

PROGRAM TESTSET;
VAR
  DIGITS,LETTERS,LOWERCASE,UPPERCASE : SET OF CHAR;
  ALPHANUMERIC,ALPHA                 : SET OF CHAR;
  D : CHAR;
BEGIN
  D:='1';
  DIGITS:=['0'..'9'];
  LOWERCASE:=['a'..'z'];
  UPPERCASE:=['A'..'Z'];
  LETTERS:=LOWERCASE + UPPERCASE;
  ALPHANUMERIC:=LETTERS + DIGITS;
  ALPHA:=ALPHANUMERIC - DIGITS;
  IF ( D IN ALPHANUMERIC * DIGITS ) THEN
    WRITELN('PUNT');
END.
```

(C) copyright 1981 Alcor Systems

Appendix

On diskette there is a file named DATABASE/PCL . This source program ties all of the previous program segments in chapters 9 and 10 together, to build a program that will build a data base for business customers. This is not intended to be a comprehensive program, but can serve as a starting point for an expansion. This program requires approximately 15K of stack to RUN or execute. Once compiled, it may be executed by typing:

```
RUN DATABSE/PCL 15K
```

The number of customers allowed in the data base array is set by the constant "maxarray" , and may be changed to reflect local memory restrictions. Customer transactions are linked to each customer record by dynamic management of linked lists. Customer records are kept on a separate file from the transactions in order to simplify rebuilding of the linked lists when loading an existing data base. The size of the data base accessible during a program invocation is limited by the available memory, as the entire data base is loaded into memory for operations. Large data bases may be accessed by partitioning the data base between files and running the program multiple times.

Pascal Reference Manual
Alcor Systems

Second Edition
First printing
1982

Copyright (C) 1981 by Alcor Systems

SECOND EDITION
FIRST PRINTING-1982

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from Alcor Systems. No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this book, Alcor Systems assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

TRS80 and TRSDOS are trademarks of Tandy Corporation.
USCD Pascal is a trademark of UC Regents, San Diego campus.
Apple is a registered trademark of Apple Computer Inc.
Z-80 is a trademark of Zilog corporation.
CPM is a trademark of Digital Research.

TABLE OF CONTENTS

(C) copyright 1981 Alcor Systems

Chapter 10

FOREWORD

This manual assumes that the reader is already somewhat familiar with the Pascal language. It is organized to be used as a reference manual. As such, the chapters group related topics in order to make them easier to find. The result of this is that the manual does not follow a progression of discussion which is well suited as a teacher of the Pascal language. It is suggested that you first read the Pascal Tutorial if this is your first experience with the language.

NOTATION AND TERMINOLOGY

The description of any programming language involves both the syntax and the semantics of the language. The syntax refers to the arrangement of program elements into a form which the compiler can understand. The semantics refers to the meaning that the compiler associates with a particular arrangement of the program elements. The semantics of a language can be explained with words but the syntax is best explained through the use of diagrams.

The syntax diagrams used throughout this manual describe the legal syntax of a program. Each diagram has an entering and an exiting point which is denoted by an arrow. Starting with the arrow entering a diagram, the legal syntax can be determined by tracing a path which follows the directions indicated by the arrows until the exiting arrow is reached. Most diagrams have a multiple number of paths from starting point to ending point. All paths describe a syntactically correct form.

The following are sample syntax diagrams which describe the syntax of an integer.

Syntax of an integer:

```
       --> - --
      |        |  ----------
      |--> + --| |          |
      |        V V          |
  ---------------> digit --->
```

Syntax of a digit:

```
       --> 0 ---
      |         |
      |--> 1 ---|
      |         |
      |--> 2 ---|
      |         |
      |--> 3 ---|
      |         |
      |--> 4 ---|
      |         |
      |--> 5 ---|
      |         |
      |--> 6 ---|
      |         |
      |--> 7 ---|
      |         |
      |--> 8 ---|
      |         V
  ------> 9 ------>
```

The syntax diagram for an integer says that an integer is a concatenation of one or more digits which is optionally preceded by a plus or minus sign. Entering the diagram, you have 3 possible paths from which to choose. One path leads directly to "digit", one leads to "+", and one leads to "-". The paths from both "+" and "-" then lead to "digit". Passing through "digit", you have the option of exiting the diagram or following the arrow which leads back to the beginning of "digit". From this point, you pass through digit again and optionally exit or return for another pass. Thus, an integer may consist of one or more digits.

The second syntax diagram describes the correct forms of a digit. Entering the diagram, you have ten possible paths from which to choose. All paths lead to a single character, each of which is a legal digit. Choosing a path, you follow it through a character and end up at the exiting arrow. At this point, there is no alternative but to exit the diagram. No other paths are available. Some examples of integers then are 10, +963, and -75.

In the diagrams used in this manual, upper case character strings denote reserved words that must be present in the form shown. Lower case character strings denote the parts of the syntax where many legal forms exist. For example, the word integer in a diagram in lower case letters represents any legal integer. The word INTEGER in uppercase letters represents a reserved word of the language.

In some cases, abbreviations are used to shorten a diagram. For example, id is used in place of identifier. Also, expr is used in place of expression. A few other abbreviations may occur but where used, their meaning should be apparent from the surrounding text.

PROGRAM ELEMENTS

The elements of a program consist of the entities (identifiers, numbers, strings, reserved words, and special symbols) which are composed from a character set. The ASCII character set is the most often used and is listed in the appendix.

A. Identifiers

An identifier serves to denote the program name, a constant, a type, a variable, a procedure, or a function. It consists of a letter followed by combinations of zero or more of the following characters:

(the 26 letters of the alphabet in lower or upper case, the digits 0 through 9, the character $, the character _).

Note:
no distinction is made between upper and lower case letters in identifiers. The two identifiers, Apple and apple, are considered identical.

The length of an identifier is arbitrary but only the first 8 characters are significant. For example, the identifiers A2345678 and A23456789 would to the compiler be identical because it discards all characters past the eighth character. Therefore, care should be taken to make identifiers eight characters unique. It should also be noted that an identifier cannot contain embedded blanks or span a line boundary.

Examples: Factor$ DEPARTMENT A Div_10 B12345678$_

(C) copyright 1981 Alcor Systems

B. Numbers

Numbers are integer or real constants. Integers are allocated sixteen bits of storage which imposes a size limitation. The range for an integer is -32768 to +32767.

Syntax of integer numbers:

```
        --> - ---
       |         |
       |--> + ---|      ----------
       |         |     |          |
       |         V     V          |
  --------------------> digit --->
```

Examples of integer numbers:

30 -28934 0 32739

Real numbers are represented in either scientific or floating point form. The floating point form consists of an integer part followed by a decimal point and a fractional part. The scientific form consists of a floating point part followed by an exponent part. The exponent part is a multiplier. The value of a real number in scientific form is the floating point part times (10 raised to the exponent part). (See the System Implementation Manual for the size, range, and accuracy of real numbers).

Syntax of real numbers:

```
                       ----------    -----------------------
                      |          |  |                       |
                      V          |  |                       V
--> integer ---> . ---> digit -----------> E ----> integer --->
                                   |         ^
                                   |         |
                                   --> D ----
```

Examples of real numbers:

Floating point form:

50.0 -100000.0 345.22452

Scientific form:

0.239E3 -4.5921E-2 876.0E+99 193.27D-3

0.239E3 is equivalent to 239.0
-4.5921E-2 is equivalent to -0.045921

NOTE: Using D instead of E in scientific form represents a double precision real number.

(C) copyright 1981 Alcor Systems

C. Strings

Strings are sequences of characters enclosed by single quote marks. A string consisting of a single character is a constant of the type CHAR. Strings consisting of n characters, where n is greater than one, are constants of the type PACKED ARRAY[1..n] OF CHAR. If a string is to contain a single quote mark, it must appear twice in the sequence.

Examples: 'ABC' '12"QZW' 'BEGIN ' '***' ' %'

The string consisting of the single character ' is represented as ''''.

Characters in strings can also be denoted by hexadecimal numbers. A hexadecimal number is composed from the characters 0 through 9 and A through F. (See the ASCII character set in the appendix). The character # followed by 2 hexadecimal characters represents a single ascii character. The character represented is the one whose ordinal position in the character set corresponds to the hexadecimal number specified. This feature provides a mechanism for representing nonprintable characters. A consequence of giving the character # a special meaning is that it must appear twice in a string just as the character ' must when the character itself is to be made a part of the string. A string consisting of the single character # then is represented by '##'.

Examples of hexadecimal character representation in strings:

```
'#30'   is equivalent to '0'
'D#4FG' is equivalent to 'DOG'
'#00'   corresponds to the nonprintable null character
'A#B'   is illegal
```

D. Reserved Words

The following list of words are keywords and have special meaning in a program. They may not be used as identifiers.

AND	DOWNTO	IF	OR	THEN
ARRAY	ELSE	IN	PACKED	TO
BEGIN	END	LABEL	PROCEDURE	TYPE
CASE	FILE	MOD	PROGRAM	UNTIL
CONST	FOR	NIL	RECORD	VAR
DIV	FUNCTION	NOT	REPEAT	WHILE
DO	GOTO	OF	SET	WITH

(C) copyright 1981 Alcor Systems

E. Special Symbols

The special symbols are used as operators and delimiters in a program. Because character sets vary from system to system, alternate representations are provided for some of the symbols.

Symbols with only one representation:

```
+     -     *     /
=     <>    <     <=    >=    >
(     )     '     :=    .     ,
;     :     #     ::
```

Symbols with alternate representations:

symbol	alternate
{	(*
}	*)
^	@
[	(.
]	.)

F. Comments

Comments can be used in a program for documentation purposes. The compiler generates no code for comments. The symbol { denotes the the beginning of a comment while the symbol } denotes the end. All characters in between are ignored by the compiler. As shown above, the symbol { may be replaced by the symbol (* and the symbol } may be replaced by the symbol *).

Examples:
```
{this is a comment}
(*This is a comment
  that spans more than one line*)
```

Note: Comments may not be nested. The following will generate an error:

```
(*outer (*inner level*) level*)
```

G. The Semicolon

The semicolon is used extensively in the Pascal language. Its purpose is to separate the individual components of a program. For example, block headings must be separated from block parts, block parts must be separated from one another, and individual definitions, declarations, and statements within the block parts must be separated. In general, they may be used freely throughout the program. However, care should be taken not to include a semicolon in the middle of a statement. This is a common source for error when using the IF statement with one or more ELSE clauses. Since the ELSE clauses are a part of the IF statement, they must not be separated from it by a semicolon. An ELSE keyword should never be preceded by a semicolon.

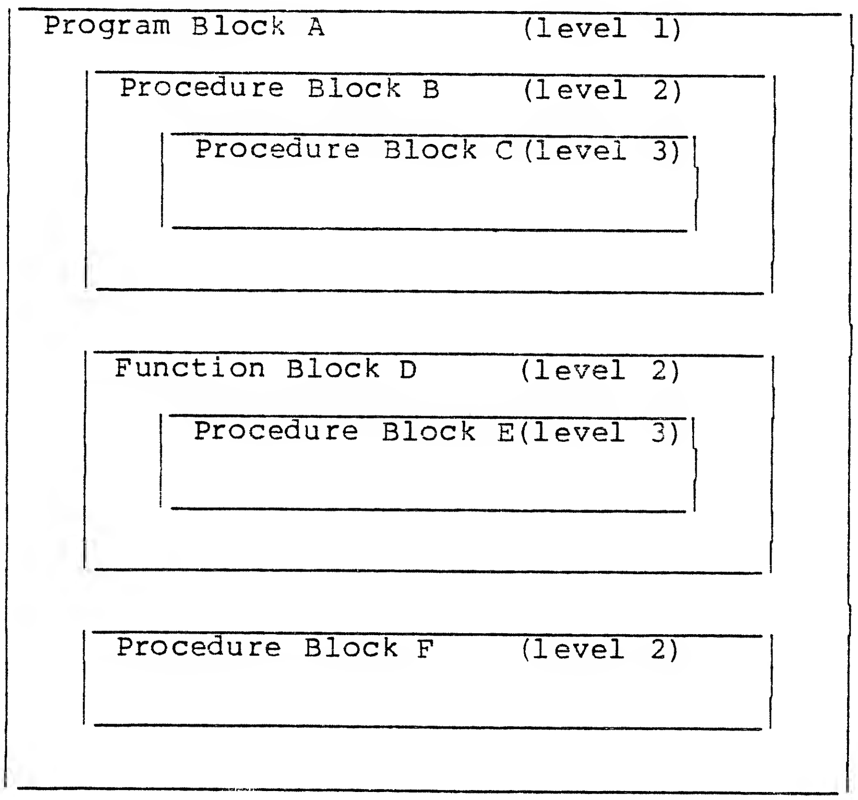
example use of semicolons in an IF statement:

```
IF time > 12 THEN
  BEGIN
  alpha := 'e';
  beta  := 'f';
  END          (*semicolon here would cause an error*)
ELSE
  BEGIN
  alpha := 'g';
  beta  := 'h';
  END;
```

(C) copyright 1981 Alcor Systems

PROGRAM STRUCTURE

Pascal is a block structured language. This means that a program is constructed in a block like manner. At a minimum, a program consists of one block. More blocks are created through the use of procedures and/or functions by placing them inside this outermost "program block". The term for this process is called nesting. The rule for nesting is that a block may lie entirely within another block, but blocks do not overlap in any other way. A level of nesting can be assigned to each block of a program. This provides an appropriate tool for describing scope rules which are discussed in chapter 9. The block structured organization of a program can be represented pictorially by the following example:



A program then consists of at least one block, the program block, and optionally it contains procedure and/or function blocks which are nested within.

(C) copyright 1981 Alcor Systems

A. Block Headings

The purpose of the block heading is to give the block a name and in the case of procedure or function blocks, to define any parameters to be passed to the block. There are three types of blocks: the program block, the procedure block, and the function block. There is only one program block, the outermost block of the program, while there may be any number of procedure and function blocks. Each of the three types of blocks has a different heading. (Procedures and functions are discussed further in chapter 9)

A.1 The Program Heading

The program heading must be the first non-comment in a program. Its purpose is to signal the start of the program and to give the program a name. Characters inside the parentheses are ignored by the compiler.

Syntax of the program heading:

```
                         -------------------------
                         |                       |
                         |                       V
 --> PROGRAM  --> id ---> ( --> comments --> ) ---> ; -->
```

Example program headings:

```
PROGRAM lander;        PROGRAM taxes(computes income tax);
```

A.2 The Procedure Heading

The procedure heading signals the start of a procedure block. It gives the procedure a name and defines the parameters to be passed to it.

Syntax of the procedure heading is:

```
                             --------------------
                             |                  |
                             |                  V
 --> PROCEDURE --> id ---> parameter list  ---> ; -->
```

(C) copyright 1981 Alcor Systems

The parameter list declares the variables which are used to pass data into and out of a procedure. The variables are called formal parameters. The procedure statement which activates (or calls) a procedure has a corresponding list of parameters which are the actual parameters. The actual parameters must match the formal parameters in order and in type. However, their names need not be the same.

There are two different kinds of formal parameters, pass by value or pass by reference. A formal pass by value parameter causes its corresponding actual parameter to be copied to another location and then the formal parameter references the copied value. Therefore, changing the value of the formal parameter inside the procedure does not change the value of the corresponding actual parameter. In contrast, a formal pass by reference parameter is passed the address of the corresponding actual parameter. The formal parameter references the same location as the actual parameter. Therefore, changing the value of the formal parameter also causes the value of the actual parameter to be changed. Variable declarations in the parameter list which are preceded by the keyword VAR are pass by reference parameters while the absence of the keyword represents pass by value.

Syntax of the parameter list is:

```
           -------------------- ; <--------------------
          |                                             |
          |  -------    ---- , ----                     |
          | |       |  |          |                     |
          V |       V  V          |                     |
--> ( ------->VAR -------> id -------> : --> type id -------> ) -->
```

Example procedure headings:

```
PROCEDURE out;

PROCEDURE cpu(pc : INTEGER);

PROCEDURE delete(VAR i,j :INTEGER; ch :CHAR; VAR x :REAL);
```

In procedure delete above, i and j are integers which are passed by reference, ch is a character which is passed by value, and x is a real which is passed by reference.

As a general rule, pass by value parameters should be used to prevent side affects. However, sometimes side affects are necessary. That is, sometimes you need a change in the value of a formal parameter to also change the value of its corresponding actual parameter. In such a case, pass by reference must be used. Also, when passing large data structures such as arrays, pass by reference should be used. This speeds execution and saves memory because a pointer to the structure is passed rather than copying the whole structure to another location.

A.3 The Function Heading

The function heading signals the start of a function block. It gives the function a name and defines the parameters to be passed to it. Unlike a procedure, a function has a type associated with it. Functions, like variables, are assigned values. A function is referenced by an expression and its value then substituted into the expression.

Syntax of the function heading:

```
                           -------------------
                          |                   |
                          |                   V
--> FUNCTION --> id ---> parameter list ---> : --> type id --> ; -->
```

The parameter list has the same form as the parameter list for a procedure discussed on the previous page.

Example function headings:

```
FUNCTION number : REAL;

FUNCTION nextstate(currentstate : INTEGER) : INTEGER;
```

The function "number" is a real valued function which has no parameters. The function "nextstate" is an integer valued function which has one parameter, also of type INTEGER. In each function, a value should be assigned to the function name. For example, number:=5.3 and nextstate:=currentstate + 1 could appear inside each of the respective functions to define values for them.

(C) copyright 1981 Alcor Systems

B. Block Parts

A Block is composed from the following list of parts.

1. the label declarations
2. the constant definitions
3. the type definitions
4. the variable declarations
5. the common declarations
6. the access declarations
7. the procedure and function declarations
8. the statement body

The label declarations are used to declare statement labels which can be used for branching. The constant definitions are used to give names to numbers or strings which are constants. Constant names are assigned values at compile time. Type definitions are used to create and give names to data types which are not predefined. Variable declarations are used to associate variable names to specific data types. A type defines the kind of data that can be stored in a variable. It also defines the amount of storage required for the variable. Variables are assigned values at run time. Common declarations are used in the same manner as the variable declarations to associate variable names to specific data types, but common variables have a special property. Storage space for common variables is created statically rather than dynamically. This means that when a block terminates, the common variables declared in it do not become undefined. Access declarations are used to enable a block to access a common variable. Procedures and functions are used for modularity. They provide the mechanism for segmenting a block into subblocks. The statement body contains the program statements which describe the actions to be taken on data as well as the order in which the actions take place.

A block does not have to include all eight parts described above. At a minimum, a block must include the two keywords BEGIN and END which bracket the statement body. The following is an example of a minimum complete program. It contains only the program block which is composed of only the heading and a null statement body.

```
PROGRAM donothing;
BEGIN           (*The statement body contains no statements*)
END.
```

(C) copyright 1981 Alcor Systems

The order in which the eight parts appear in a block is as follows: The first six parts may be arranged in any order. The only requirement is that an identifier be defined before it is used. For example, a particular type definition must textually precede a variable declaration of that type. The only exception to this is the definition of pointer types which are discussed in chapter 5. It is also worth noting that there may be more than one of a particular part. For example, there could be two separate type definition parts. The procedure and function declarations follow any use of the first six parts. The statement body then follows the last procedure or function declaration.

B.1 The Label Declarations

Label declarations are used in conjunction with the GOTO statement. A label declaration defines a label which can then be used to label a statement. A GOTO statement can then reference the label causing a branch to the statement which is prefixed by the corresponding label. The label declaration part is signaled by the keyword LABEL.

Syntax of the label declaration part:

```
                 --------- , <--------
                 |                   |
                 V                   |
--> LABEL ---> integer constant ---> ; -->
```

Note:
A label must be declared in the same block in which a GOTO statement which references it appears. Branching outside a block is not allowed. Also, all declared labels must appear somewhere in the statement body.

Example label declaration part:

```
LABEL 100, 200, 300, 400, 500, 1000 ;
```

Syntax of labeled statement:

```
---> LABEL --> : --> statement --->
```

Example labeled statements:

```
100: x:=47;
200: IF x > 500 THEN GOTO 100;
```

(C) copyright 1981 Alcor Systems

B.2 The Constant Definitions

The constant definitions are used to associate identifiers with values which do not change. A constant identifier is assigned a value at compile time and this value can not be changed. This means that a constant identifier cannot have its value changed by an assignment statement. The use of constant identifiers increases program readability because meaningful names can be used in the place of actual values. The values which can be assigned to constant identifiers are numbers, strings, or other identifiers which are constants. This includes identifiers which are members of an enumeration. The start of the constant definition part is signaled by the keyword CONST.

Syntax of the constant definition part:

```
                 ------------ ; <----------
                !                          !
                V                          !
  --> CONST ---> id --> = --> constant ---> ; -->
```

Example constant definition part:

```
CONST     low=32;  high=88;  pi=3.14159;
          speedoflight=299792.0;  separator='---------';
          positive=10;  negative=-positive;
          keydefinition=#61;
```

Note: Integer constants may also be expressed in in hexadecimal by preceding the value with the #

There is a predefined constant MAXINT which is defined to be equal to the largest positive value an integer can take.

(C) copyright 1981 Alcor Systems

B.3 The Type Definitions

Type definitions are used to create new data types. A type definition associates a name with a user defined simple or structured data type. The name can then be used in a variable declaration to specify the type of the variable. Although a variable can declare its type directly in the variable declaration part, it is nice and sometimes necessary to have a name associated with a user defined type. Type definitions are especially useful when using structured types whose definitions are long and when more than one variable in the program is to be declared of that type. Associating a name to the type means that the type must be defined only once. In some cases, type definitions are necessary. If comparisons are to be made between two variables of a user defined type, then the variables must be declared as the same type. Defining the type for each variable separately in a variable declaration part will not work. Although the variable declarations will look the same, the compiler will view them as variables of two separate types. Also, declarations of variables in the parameter list of a procedure or function must be to named types. For example, if an array is to be passed as a parameter, the array must be defined in a type definition and then the formal parameter declared as that type.

The type definitions part is signaled by the keyword TYPE.

Syntax of the type definition part:

```
                  --------- ; <---------
                 |                      |
                 V                      |
--> TYPE ---> id --> = --> type ---> ; -->
```

Example type definition part:

```
TYPE     colors = (red,blue,green,orange,purple);
         weekdays = (sunday,monday,tuesday,wednesday,
                     thursday,friday,saturday);
         workdays = monday..friday;
         daysofmonth = 1..31;
         letters = 'A'..'Z';
         list = ARRAY [0..25] of CHAR;
         customer = RECORD
                    name    : PACKED ARRAY[1..20] OF CHAR;
                    address : PACKED ARRAY[1..40] OF CHAR;
                    END;
```

(C) copyright 1981 Alcor Systems

B.4 Variable Declarations

All variables in a program must be declared before they are used. This is done by associating the variable name with a type. The type can be the name of a predefined simple data type, the name of a user defined type which has previously been defined in a type definition part, or the type can be defined directly. The start of the variable declaration part is signaled with the keyword VAR.

Syntax of the variable declaration part:

```
                 ------------ ; <------------
                |                            |
                |   --- , <--                |
                |  |         |               |
                V  V         |               |
   --> VAR ------> id -----> : --> type ---> ; -->
```

Example variable declaration part:

```
VAR      x,y,z        : REAL;
         ok           : BOOLEAN;
         i,j,k        : INTEGER;
         fruit        : colors;
         alpha,beta   : CHAR;
         characters   : list;
         mark         : ARRAY [1..30] OF INTEGER;
         byte         : 0..255;
         months       : (jan,feb,mar,apr,may,jun,jul,
                         aug,sep,oct,nov,dec);
         account      : RECORD
                        number, date : INTEGER;
                        END;
```

B.5 Common Declarations

The common declaration part is used in the same manner as the variable declaration part to associate a variable name with a specific data type. However, declaring a variable in the common declaration part gives it a special property. Normally, storage space for variables is allocated dynamically. This means that the variables declared in a block are allocated storage space when the block is activated and the space is freed when the block terminates. Therefore, the local variables of a block become undefined when the block terminates. In contrast, common variables are allocated storage space statically at compile time. This means that the common variables of a block retain their defined values even after the block terminates.

(C) copyright 1981 Alcor Systems

Common variables are scoped just the same as normal variables but a common variable cannot be accessed in a block unless its name appears in an access declaration of the same block. This feature is useful for controlling access to global variables, providing protection and better documentation of where global variables are used. Another very valuable use for common variables is in external procedures. A procedure which is often used by many separate programs can be compiled separately and linked to the programs that use it. In the case where the procedure must retain information between activations, such as cursor position in a graphics procedure, common variables may be used to prevent the need for declaring the necessary variables globally in every program that uses the procedure.

Syntax of the common declaration part:

```
             ----------- ; <-------------
            |                             |
            |  ---  , <--                 |
            | |          |                |
            V V          |                |
--> COMMON -----> id -----> : ---> type ---> ; --->
```

Example common declaration part:

```
     COMMON          cursorx , cursory  : INTEGER;
```

B.6 Access Declarations

Access declarations are used in conjunction with common variables. No common variable can be referenced unless its name appears in an access declaration of the block which references it. The order in which common variable names appear in an access declaration is arbitrary.

Syntax of the access declarations part:

```
               ---  ,  <--
              |          |
              V          |
--> ACCESS ------> id ------> ; --->
```

Example access declaration part:

```
     ACCESS       cursorx , cursory;
```

B.7 Procedure and Function Declarations

Procedure and function declarations create new blocks. Each declaration forms a complete new block composed from the block parts discussed earlier. A procedure declaration consists of a procedure heading followed by a block. A function declaration consists of a function heading followed by a block. Procedure and function declarations form subblocks within the block in which they appear. Procedure and function declarations are discussed more fully in chapter 9.

Syntax of procedure or function declaration:

```
              --> function heading --
             |                       |
             |                       V
        -----> procedure heading ----> block --->
```

Example procedure declaration:

```
 PROCEDURE getvalue(first,last :INTEGER; VAR word : buffer;
                                         VAR value: INTEGER);
(*Converts hex character string to decimal value:
  buffer is a globally declared type --> PACKED ARRAY[1..8] OF CHAR;
  word contains the hex character string
  first and last are pointers into the string
  value is the returned decimal value                              *)

VAR  i,n,factor      : INTEGER;
     ch              : CHAR;

BEGIN
  value := 0; factor := 1;
  FOR i := last downto first DO
    BEGIN
    ch := word[i];
    IF ch = ' ' THEN n:=0     (*Blank character given value 0*)
    ELSE
      IF (ch>='0') AND (ch<='9') THEN   (*character range 0..9 *)
        n := ORD(ch)-ORD('0')           (*convert ch to decimal*)
      ELSE
        IF (ch>='A') AND (ch<='F') THEN (*character range A..F *)
          n := ORD(ch) - ORD('A') + 10; (*convert ch to decimal*)
    value := value + factor * n;
    factor := 16 * factor;             (*hex is base 16*)
    END;
END;                                   (*procedure getvalue*)
```

Example function declaration:

```
FUNCTION nextstate(currentstate : INTEGER) : INTEGER;
(* returns the next state given the current state *)

BEGIN
  CASE currentstate of
    1: nextstate := 3;
    2: nextstate := 4;
    3: nextstate := 1;
    4: nextstate := 2;
  END;
END;                                  (*function nextstate*)
```

B.8 Statement Body

The statement body of a block contains zero or more statements which describe the actions of the block. The statement body must start with the keyword BEGIN and stop with the keyword END. However, since statements may also include BEGIN and END, the statement body may contain many occurrences of these two keywords. The statement bodies for the three types of blocks are identical, except that the concluding END for the program block statement body must be followed by a period while the concluding END for procedure and function statement bodies must be followed by a semicolon.

Syntax of statement body:

```
                                              --> ; --
                                             |        |
                                             |        V
       ---> BEGIN --> statements --> END ----> . ----->
```

Example statement body:

```
BEGIN          (* begin program block statement body *)
  WHILE NOT EOF DO
    BEGIN
    READ(x,y,z);
    x := SQR(x); y := SQR(y); z := SQR(z);
    WRITE('squaredata ' , x , y , z);
    END;
END.           (* end of program *)
```

SIMPLE DATA TYPES

The simple data types are the primitive data types of the language. They form the base for building structured types. The simple data types consist of ordinal types and the REAL type.

A. Ordinal Types

Ordinal types are characterized by a linear ordered set of distinct values which can be mapped on the set of natural numbers. This mapping is actually an enumeration of all the values which the type can take. The predefined ordinal types are INTEGER, CHAR, and BOOLEAN. New ordinal types can be defined by enumerating all the values which the type can take. In addition, new ordinal types may be defined as subranges of other ordinal types.

A.1 The Type INTEGER

Variables declared as type INTEGER may take on values in the range -32768 to +32767. All the arithmetic and relational operators can be used with integer constants and variables. However, the relational operator IN is used only in conjunction with sets (see chapter 4).

Syntax of type INTEGER:

```
--> INTEGER -->
```

Example declaration:

```
VAR          i,j,k : INTEGER;
```

Example integer constants:

```
59    -1    0    329    -10000    29872
```

(C) copyright 1981 Alcor Systems

A.2 The Type CHAR

Variables declared as type CHAR can take single characters as values. The set of valid single characters is defined by a character set. All characters have an associated ordinal number in the range 0 to 255. A table of ASCII characters with associated ordinal numbers is listed in the appendix. There are two functions which may be used in conjunction with the character set. The function ORD(character) returns the ordinal number of the character specified. The function CHR(ordinal number) returns the character associated with the specified ordinal number. These are known as transfer functions because they are used to transfer a character value to an integer value and vice versa. Constants of type CHAR are denoted by using single character strings. All relational operators may be used with variables and constants of type CHAR.

Syntax of type CHAR:

```
--> CHAR -->
```

Example declaration:

```
VAR        alpha , beta    : CHAR;
```

Example character constants:

```
'9'   'a'   '#9F'
```

Example relational expression:

```
'A' < 'B'
```

A.3 The type BOOLEAN

The boolean type represents logical data. A logical value is represented by the predefined identifiers FALSE and TRUE. These are the only possible values of a boolean variable or expression.

Syntax of type BOOLEAN:

```
--> BOOLEAN -->
```

The boolean type is defined by the following enumeration:

```
BOOLEAN = (FALSE, TRUE)
```

The boolean operators AND, OR, and NOT take boolean operands and yield boolean results. The relational operators = , <> , <= , < , > , >= , and IN all yield boolean results. See chapter 7 for examples of boolean expressions.

Example declaration:

```
VAR  switch  : BOOLEAN;
```

Boolean Constants:

```
FALSE    TRUE
```

A.4 The Enumerated Type

Pascal allows you to define your own ordinal types. A new type may be created by enumerating all the values that the type may take. This is done by giving the new type a name and listing the values which the new type can take.

Syntax of the enumerated type:

```
                   --- , <--
                  |         |
                  V         |
          --> ( ----> id ----> ) -->
```

Example definitions of enumerated types:

```
names = (Fred, Joe, Nancy, Susan);

foods = (hotdog, hamburger);
```

The values listed are identifiers. The order in which the identifiers are listed defines a relationship. The identifiers can be thought of as being mapped on to a set of natural numbers. The first identifier maps to 0, the second to one, the third to two, and so on. This implies that identifier1 < identifier2 < identifier3...< identifierN. For example, consider the predefined type:

```
BOOLEAN = (FALSE,TRUE)
```

The boolean value FALSE is less than the boolean value TRUE because FALSE appears in the list before TRUE. This kind of ordered relationship applies to any enumerated type. Consider the type definition:

```
colors = (red, blue, green)
```

By this definition, a variable declared as type colors can take on the the value red, blue, or green. The definition also implies that red < blue < green.

The ordering means that enumerated values can be used in relational expressions. It also means that they may be used for range specifications. For example, consider the FOR statement. The range of the loop control variable is defined by specifying a starting and stopping value. These starting and stopping values could be the values of an enumerated type. For example, if color has been declared as type colors, the following statement is valid:

```
FOR color := red to green DO .....
```

(C) copyright 1981 Alcor Systems

A.5 Subrange Types

A subrange type is simply a type defined to take on a subset of the values representing some ordinal type.

Syntax of the subrange type:

--> constant --> .. --> constant -->

The use of subranges can sometimes save memory. For example, an integer variable whose values are always in the range of 0 to 255 could be declared as a subrange of the type INTEGER. You might define a new type as follows:

```
byte  = 0..255
```

Now, variables declared as type byte would be allocated 8 bits of storage rather than the 16 bits which is allocated for variables declared as type INTEGER. The compiler allocates the minimum amount of storage required to represent the range of values specified by a subrange type.

The use of subranges can better document a program by defining the range of valid values a variable declared as the subrange type can take on. Subrange types are also often used in conjunction with SET types which are discussed in section B of chapter 4.

B. The Type REAL

The type REAL is used to represent fractional numerical data. The implementation of reals is machine dependent. Information on the size, range, and accuracy of reals is discussed in the System Implementation Manual. See section B of chapter 1 for the syntax of real constants.

Syntax of REAL:

--> REAL -->

Example declaration:

```
VAR      x , y , z    : REAL;
```

(C) copyright 1981 Alcor Systems

STRUCTURED DATA TYPES

There are four kinds of structured data types: the ARRAY, the SET, the FILE, and the RECORD. These four kinds of data types represent four different ways of organizing the simple data types into a data structure. A data structure can also include other data structures as components. It is then possible to build very complex structures from the basic simple data types. All structured data types can be packed. This means that the most compact form of storage possible will be used. Packing a data structure can sometimes save memory. However, packing may cause access time to increase. The decision to pack or not depends on the specific application. The keyword PACKED signals the compiler to pack the data type into its most compact form. When structured types contain other structured types, the keyword PACKED must be applied to the innermost structure as well as the outermost to have any effect.

A. The Type ARRAY

The ARRAY is a data type which defines a structure composed of a fixed number of data elements which are all of the same type. The data elements can be defined to be of any one type. They could be defined as one of the simple types or as one of the structured types, including ARRAY. Arrays can be defined to be of any dimension. The number of dimensions, the number of elements in each dimension, and how the elements are accessed is specified by an index definition. The index definition consists of a list of ordinal types(excluding the type INTEGER for the reason that this would create an array too large to fit in memory). The number of types specified corresponds to the number of dimensions of the array.

Syntax of the type ARRAY:

```
   -----------                       ------ , <-------
  |          |                      |                 |
  |          V                      V                 |
---> PACKED ---> ARRAY --> [ ---> ordinal type ---> ] --> OF --> type -->
```

(C) copyright 1981 Alcor Systems

Example declarations:

```
TYPE    table = ARRAY [0..5,1..10] OF INTEGER;
        colors = (red, blue, green, yellow);
VAR     report : ARRAY [1..20] OF table;
        day    : ARRAY [1..365] OF REAL;
        class  : ARRAY [0..8,0..5] OF INTEGER;
        chart  : ARRAY [colors] OF INTEGER;
```

Elements of variables declared as type ARRAY are accessed by specifiying the variable name and listing expressions which evaluate to ordinal values that fall into the range of the ordinal types of the index definition.

Examples of accessing array elements:

```
report[5,3,6]      day[40]      class[0,0]      chart[red]
```

B. The Type SET

A set is a collection of distinct elements which are all of the same ordinal type. The elements of a set are called set members. There may be up to 256 members in a set. The 256 member limit causes the restriction that a set can not be defined to be of ordinal type INTEGER. Also, subranges of type INTEGER which include negative integers are not allowed as set base types. A set can have no members in which case it is called an empty set.

Syntax of the type SET:

```
--> SET --> OF --> ordinal type -->
```

Example declarations:

```
TYPE    days = (sunday, monday, tuesday, wednesday,
                            thursday, friday, saturday);

VAR     lowercase, digits, special : SET OF CHAR;
            schooldays, workdays   : SET OF days;
                       day         : days;
```

A variable declared as type SET can take on any values which are subsets (including the empty set) of the values defined by the type of the set. The type of the set is specified after the keyword OF.
Set values are denoted by listing set members within square brackets. The individual members can be specified as ordinal expressions.

(C) copyright 1981 Alcor Systems

Syntax of set notation:

```
              ----------------------------------------
             |                                        |
             |  -------------- , <-----------------   |
             | |                                   |  |
             | |             ------------------    |  |
             | |            |                  |   |  |
             | V            |                  V   |  V
--> [ -----> ord expr ---> .. --> ord expr ------> ] -->
```

The .. notation between two members specifies that all values in between are also to be included as members. For example, [0..3,7..10] would denote a set with members 0,1,2,3,7,8,9,10. The empty set is denoted by [].

Example assignments to set variables:

```
schooldays:= [monday, wednesday, friday];
workdays   := [monday..friday];
lowercase  := ['a'..'z'];
digits     := ['0'..'9']
special    := ['*' , '%' , '@']
```

The relational operators which are applicable to sets are (IN , = , <> , <= , and >=)

IN

A single element can be tested to see if it is a member of a set. The operator IN is used for this testing of set membership. This operation evaluates to TRUE if the single element on the left is a member of the set on the right.

=

Two sets can be compared to see if they contain exactly the same members. The operator = is used to test for set equality. If each member of each set is also a member of the other then the operation evaluates to TRUE.

<> Two sets can be compared to see if they do not contain exactly the same members. The operator <> is used to test for set inequality. If any member of either set is not also a member of the other then the operation evaluates to TRUE.

<=

A set can be compared to another set to see if the first set is a subset of the second set. The operator <= is used to test for set inclusion. If all the members of the set on the left are also members of the set on the right then the operation evaluates to TRUE.

>=

A set can be compared to another set to see if the first set is a superset of the second set. The operator >= is used to test for set containment. If there are no members in the set on the right which are not also members of the set on the left then the operation evaluates to TRUE.

Example use of relational operators:

```
IF day IN workdays THEN gotowork; (*gotowork is a procedure*)
IF character IN digit THEN WRITE(character);
IF workdays >= schooldays THEN noweekendclasses;
```

Relational Expression	Evaluation
monday IN [monday, tuesday]	TRUE
'A' IN ['a'..'z']	FALSE
[1, 2, 3] >= [0]	FALSE
[1, 2, 3] >= [2]	TRUE
['%'] <= ['*', '%']	TRUE
[] <= [tuesday]	TRUE
['a', 'f', 'g'] = ['a', 'f', 'k']	FALSE
[1] = [1]	TRUE

The arithmetic operators which are applicable to sets are (+ , - , and *).

\+

Two sets can be combined to form a third set containing all elements that are members of either set. The operator + performs the union of two sets.

\-

A set can be formed as the difference between two sets. The operator - performs set difference. The result is a set containing all members of the set on the left which are not also members of the set on the right.

* A set can be formed which contains only the members which exist in both of two other sets. The operator * performs the intersection of two sets.

Examples:

Expression	Result
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
[1, 2, 3] + [2, 3, 4]	[1, 2, 3, 4]
[1, 2, 3] - [2]	[1, 3]
[1, 2, 3] - [4]	[1, 2, 3]
[1, 2, 3] * [4, 5, 6]	[]
[1, 2, 3] * [2, 3, 4]	[2, 3]

C. The Type FILE

The data type FILE provides the link between a program and the peripheral equipment of the computer system. Variables declared as type FILE represent logical files. Input and output operations always refer to logical files. Each logical file has an associated physical file. The physical file is the actual device to which an operation is directed. A physical file is a device such as a terminal, printer, disk file, etc... Since all input and output operations reference logical files rather than physical files, a programs input or output can be redirected simply by associating the logical file with a different physical file. The method of associating logical files to physical files is discussed in the System Implementation Manual.

File data elements can be of any type except FILE or structured types containing a component of type FILE. The use of the CLOSE procedure will assure that file data will not be lost if the program abnormally terminates and does not properly close the file. It may also be used in conjunction with the external runtime routine SET$ACNM.

(see System Implementation manual)

(C) copyright 1981 Alcor Systems

Syntax of type FILE:

```
--> FILE --> OF --> type -->
```

Input and output can be greatly simplified by declaring variables as files of structured types. For example, a complete record can be read or written to a file of records simply by specifying the file variable name and the record variable name as parameters to an input or output procedure.

Example of file declarations:

```
Type     sales = RECORD
                 salesman : PACKED ARRAY[1..20] OF CHAR;
                 quantitysold : INTEGER;
                 END;
VAR      salesfile : FILE OF sales;
         numbers   : FILE OF INTEGER;
```

The data elements of files declared as above are read and written in binary format. Binary format is the form in which the data is actually stored in memory. No translation of the data is done during the I/O process to a character readable form. The advantage of this type of I/O is speed of data transfer and minimization of disk storage requirements. The disadvantage is that the data is in a non-readable form.

A special type of file is provided for handling character formatted data. In a TEXT file, data is stored as characters. Input and output then involves a translation to and from the internal binary data format.

C.1 The type TEXT

There is a predefined type of file called TEXT. Text files have special characteristics. Unlike other file types, a text file is divided into lines. There is some mechanism which is implementation dependent which marks the separation between lines, each line being a sequence of characters. The data types which can be input from and output to text files are not restricted to characters only, even though a text file is actually a file of characters. The characters of a text file may represent string, integer, real, or boolean values. The Pascal I/O routines make the appropriate character to binary and binary to character conversions with TEXT files. There are two predeclared variables of type TEXT (INPUT and OUTPUT). These are the default parameters for the I/O procedures and functions discussed in chapter 10.

(C) copyright 1981 Alcor Systems

Example declarations:

```
infile, outfile : TEXT;
```

There are two built in procedures and one built in function which apply only to text files.

WRITELN
: The procedure WRITELN terminates the current line and positions a file pointer to the next line. If any variables are specified to be output by the WRITELN, they are output first and then the file pointer is advanced to the next line.

READLN
: The procedure READLN causes the file pointer to be positioned to the beginning of the next line. If any variables are specified to be input by the READLN, they are input first and then the file pointer is advanced to the next line.

EOLN
: The function EOLN is a boolean function which evaluates to TRUE when the end of a line has been reached. At all other times it evaluates to FALSE.

(See chapter 10 for details)

D. The type RECORD

The type RECORD is characterized by a fixed number of elements which are called fields. The fields of a record can be of different types. Record field identifiers can be declared to be of any type, including RECORD. Therefore, records can be nested.

Syntax of the type RECORD:

```
   -----------                                 ------
  |           |                               |      |
  |           V                               |      V
---> PACKED ---> RECORD --> field list ---> ; ---> END -->
```

The field list describes the individual components of a record. All the field identifiers within a record must be unique. However, field identifiers are scoped within the record itself which means that an identifier outside the record definition can be identical to a field name within the record. The field list consists of two separate parts, a fixed part and a variant part. A record can contain either or both of these two parts. If both parts are present, the fixed part must precede the variant part. The fixed part refers to the part of the record which is always referenced in the same way. (ie. the fields are fixed) The variant part refers to the part of the record which may be referenced in multiple ways. (ie. the fields may vary)

Syntax of field list:

```
                    ------------------------
                    |                      |
                    |                      V
---> fixed part ---> ; ---> variant part --->
 |                      ^
 |                      |
  ----------------------
```

Syntax of the fixed part of a record:

```
 -------------- ; <-----------
 |                            |
 |  --- , <---                |
 |  |         |               |
 V  V         |               |
------> id ------> : --> type --->
```

Example record using fixed fields:

```
RECORD
  business: PACKED ARRAY[1..25] OF CHAR;
  location: RECORD
              street,
              city,
              state    : PACKED ARRAY[1..15] OF CHAR;
              zip      : INTEGER;
            END;
END;
```

A particular field of a record variable is referenced by the variable name followed by the field name. A period separates the two names. If the field name is itself a record, then a field within the nested record is referenced by appending a period and the field name to the other two names.

Syntax of record variable referencing:

```
                          --------------------
                          |                  |
                          V                  |
--> record variable id ---> . ---> field id --->
```

Example referencing:

Assume customerrecord is defined to be of the above record type in the type definition part and that customer is declared as type customerrecord in the variable declaration part, then

```
customer.business   references first  field
customer.location   references second field
```

The nested fields of the field "location" are referenced by:

```
customer.location.street
customer.location.city
customer.location.state
customer.location.zip
```

D.1 Record Variants

Sometimes it is useful to be able to define a storage area in a record which can be accessed in multiple ways. Record variants provide the ability to do this. In certain applications, they can simplify a program and save storage space at the same time.

(C) copyright 1981 Alcor Systems

A record variant defines a fixed size storage area of a record which can be accessed in multiple ways. The size is determined by the variant alternative which requires the largest amount of storage space. The variant is defined using a form similar to that of the CASE statement.

Syntax of the variant part of a record:

```
              -----------------------
             |                       |
             |                       V
--> CASE ---> tag field id --> : ---> type --> OF ---
                                                     |
  ---------------------------------------------------
 |
 |
 |   ------------------- ; <------------------------------
 |  |                                                      |
 |  |    ------ , <----                                    |
 |  |   |              |                                   |
 |  V   V              |                                   |
  ---------> constant ---> : --> ( --> field list --> ) --->
```

Each alternative way of accessing the storage area of a variant is defined by a field list. All field names within the variant definition must be unique. The storage area can then be accessed in the desired way simply by specifying the appropriate field name. There are two forms of the variant. In one form, a tag is specified and becomes a field in the record. The tag field resides in the record just prior to the variant storage area. The purpose of the tag field is to store a value which specifies for each record the alternative of the variant which is in effect. The other form omits the tag field which in some cases is not needed.

Example using no tag field:

```
PACKED RECORD
  CASE BOOLEAN OF
    FALSE: (whole         :INTEGER);
    TRUE : (byte1,byte2   :0..255;);
END;
```

This variant definition would define a storage area of two bytes (assuming an integer is 16 bits) which is the largest amount of storage required for either of the two field lists. You could then access the whole two byte storage area as an integer or you could access each individual byte of the integer. The storage could be pictured as follows:

```
 ___________         ___________
|           |       |   bytel   |
|   whole   |  or   |___________|
|           |       |   byte2   |
|___________|       |___________|
```

The type BOOLEAN was chosen as the selector of the CASE because it defines two possible values which is what is needed to specify the two alternatives. Another type could have been defined and used just as well. With the variant defined as above, you could now reference the integer or the bytes simply by specifying the appropriate field name: whole, bytel, or byte2. For example, if "number" is a variable declared as this record type, then "number.whole", "number.bytel", and "number.byte2" are the possible ways of referencing this storage area. Care must be taken when using variants for this purpose. The way in which the fields of the different forms of the variant overlap one another is implementation dependent. Also in the above example, which byte would be the low byte and which would be the high byte is implementation dependent. (See the System Implementation Manual for information on packing)

Example using tag field:

Assume the type definition:
itemtype = (circle, rectangle, triangle)

```
PACKED RECORD
  xcoordinate, ycoordinate :REAL;
  CASE item    :itemtype OF
    circle     :(radius                    :REAL);
    rectangle  :(length,width              :REAL);
    triangle   :(baselength                :REAL;
                 angle                     :INTEGER);
 END;
```

This record definition contains a fixed part as well as a variant part with a tag field. The storage allocation for this record could assume the following structures:

```
              | xcoordinate  |      | xcoordinate  |      | xcoordinate  |
              |--------------|      |--------------|      |--------------|
              | ycoordinate  |      | ycoordinate  |      | ycoordinate  |
 tag          |--------------|  or  |--------------|  or  |--------------|
 field-->     |     item     |      |     item     |      |     item     |
              |--------------|      |--------------|      |--------------|
              |   radius     |      |  length      |      | baselength   |
              |--------------|      |--------------|      |--------------|
              |              |      |  width       |      | angle        |
              |______________|      |______________|      |______________|
```

The storage allocated would be the amount required to store the the two real numbers of the fixed part, the tag field, and the two real numbers of the rectangle field list. The other field lists of the variant require less storage than the rectangle list. The information of which alternative of the variant is in effect can now be stored as part of each record via the tag field. The tag field is referenced in the same manner as the other fields.

> Note:
> Variants can be nested. That is, a variant can contain a definition of another variant. However, there can be only one variant at any one level and the variant definiton must follow any fixed fields of a record.

POINTER DATA TYPE

The pointer data type is used in conjunction with dynamic storage allocation. This refers to the creation of storage space for variables during program execution. This is very useful when the amount of data storage a program will require is unknown. The use of pointer data types provides the ability to allocate storage as it is needed. Variables for which storage is dynamically created cannot be referenced in the usual manner. The reason is that they actually have no identifiers of their own. Instead, they are referenced through the use of pointers. A pointer is actually a variable which points to the location in memory of a dynamically created variable.

The definition of a pointer type specifies the data type for which storage will be allocated. The data type then determines the amount of storage required for each allocation. The definition of a data type does not have to precede the definiton of a pointer type which references it. This is the only exception to the rule that identifiers must be defined before they are used. This allows for a field of a record to be declared as a pointer to the record itself. Either the symbol ^ or the symbol @ may be used to signify a pointer type.

Syntax of type pointer:

```
              ---> ^ ---
             |          |
             |          V
        -------> @ -------> type id -->
```

Example pointer declarations:

```
     TYPE  transptr = @transaction;
           transaction = RECORD
                         item          :INTEGER;
                         price         :REAL;
                         link          :transptr;
                         END;
```

In the above declaration, transptr is a pointer type defined to be a pointer to the data type transaction. Transaction is a record consisting of three components (item, price, and link). Dynamic variables of the type transaction can be created through the use of pointer variables of type transptr. Notice that link is declared to be of type transptr. This component of the record is a pointer variable which may point to another dynamic variable of type transaction. Therefore, a linked list of transaction records can be formed with the link field of each record pointing to the next record.

(C) copyright 1981 Alcor Systems

The predeclared procedure NEW is used to allocate storage for dynamic variables. It has one argument which is a pointer variable. The NEW procedure allocates the amount of storage required by the data type associated with the pointer and assigns the address of the allocated storage to the pointer. The pointer is then used to reference the allocated storage. For example, consider the declaration:

```
list : transptr;
```

Then the statement NEW(list) would allocate the amount of space required to store the three components of a transaction record at some location in available memory and assign the location in memory to the variable list. The available memory is called the heap and its size is set at run time. (See the System Implementation Manual)

References to a variable which is pointed to by a pointer are made by following the pointer name with either the symbol ^ or the symbol @. In the above example, list@ would reference the dynamically created transaction record.

Syntax of referencing dynamic variables:

```
                          --> ^ --
                         |        |
                         |        V
  --> pointer id -------> @ -------->
```

Example referencing of dynamic variables:

list@	references whole record
list@.item list@.price list@.link	references individual fields
list@.link@	references record pointed to by link field
list@.link@.item list@.link@.price list@.link@.link	references individual fields

(C) copyright 1981 Alcor Systems

When a dynamically created variable is no longer needed, it may be disposed of. This is the process of freeing the space consumed by the variable for other uses. The predeclared procedure DISPOSE is provided for this purpose. Like the NEW procedure, it has one parameter which is a pointer. The DISPOSE procedure frees the memory allocated to the variable pointed to by the pointer. Referring to the above example, DISPOSE(list) would free the amount of memory which was allocated to the dynamic transaction variable.

A predefined constant NIL can be used to assign a value to a pointer. Other than using the procedure NEW, assignment to the constant NIL is the only way of giving a pointer a defined value. If a pointers value is NIL, then it does not point to a dynamic variable. This is often used with linked lists to give the pointer of the last element in the list a defined value. It provides a way of detecting when the end of the list has been reached.

Example procedures using pointer variables:

```
PROCEDURE create(VAR translist : transptr);

(* Creates a new transaction
   Adds the transaction to the top of a transaction list
   Returns a pointer to the new transaction via translist
   New transaction becomes top of transaction list*)

VAR
      trans                (*new transaction pointer*)
                  :        transptr;

      (*note: translist should be initialized to NIL*)

BEGIN
  NEW(trans);               (*create new transaction*)
  trans@.link:=translist; (*new transaction points to old top of list*)
  translist:=trans;         (*new transaction becomes top of list*)
END;                        (*procedure create*)
```

(C) copyright 1981 Alcor Systems

```
PROCEDURE   destroy(translist, trans : transptr);

(* Removes the transaction pointed to by trans from the list
   Recovers the memory used by the transaction     *)

VAR
      lead,             (*points to next transaction in list*)
      trail             (*saves location of current transaction
                          while lead is advanced to the next
                          transaction*)
                  :    transptr;

BEGIN
  lead:=translist;
  While lead <> trans DO (*search for trans*)
    BEGIN
    trail:=lead;          (*save pointer to current transaction*)
    lead:=lead@.link;     (*advance pointer to next transaction*)
    END;
  IF translist <> trans THEN (*check if trans is at top of list*)
    trail@.link:=lead@.link  (*link around transaction*)
  ELSE
    translist:=lead@.link;   (*new top of list*)
  DISPOSE(trans);            (*recover memory*)
END;                         (*destroy*)
```

(C) copyright 1981 Alcor Systems

OPERATORS

There are four categories of operators: arithmetic, relational, boolean, and type transfer.

A. Arithmetic Operators

The following table lists all the arithmetic operators, the operations they perform, the type of operands which may be used, and the type of result of the operation. Mixed mode arithmetic is supported. (eg. it is allowed to have an integer value added to a real value) Also, automatic truncation occurs when an integer variable is assigned a real value.

Operator	Operation	Type of Operands	Type of Result
+	addition	integer, real	integer, real
	set union	sets of compatible types	same type as the larger set
-	subtraction	integer, real	integer, real
	set difference	sets of compatible types	same type as the larger set
*	multiplication	integer, real	integer, real
	set intersection	sets of compatible types	same type as the larger set
/	division	integer, real	real
DIV	truncated division	integer	integer
MOD	modulus	integer	integer

Note: For sets to be of compatible types they must have identical base types , one base type must be a subrange of the other, or they may both be subranges of the same base type.

(C) copyright 1981 Alcor Systems

B. Relational Operators

All relational operators perform operations which yield Boolean results. The result is always either TRUE or FALSE. In general, both operands of a relational operator must be expressions of identical type, but the types REAL, INTEGER, and subranges of integer may be mixed.

(Relational operations may be performed on any types except files)

Operator	Result of Operation
=	true if left operand is equal to right
<>	true if left operand is not equal to right
<	true if left operand is less than right
>	true if left operand is greater than right
<=	true if left operand is less than or equal to right
>=	true if left operand is greater than or equal to right

To compare strings, the ordinal numbers of the characters composing both strings are compared to one another until a pair of characters are different or until the end of the strings is reached. If there are no character pairs which differ then the strings are equal. Otherwise, the first pair of characters which differ determine the relationship. The string whose character ordinal number is the largest is greater than the other string.

Operation	Result
´abc´ = ´cdf´	FALSE
´abc´ < ´abd´	TRUE
´bab´ > ´adf´	TRUE

The following operator tests for set membership. The left operand may be any ordinal type and the right operand may be any set of the same ordinal type.

IN true if left operand is a member of the right operand type: set
(See section B of chapter 4)

C. Boolean Operators

The boolean operators, like the relational operators yield boolean results. The result is always either TRUE or FALSE. The operands of a boolean operator must be boolean expressions.

Operator	Result of Operation
OR	true if either one or both of the operands is true
AND	true only if both operands are true
NOT	true if operand is false

Operation	Result
FALSE OR FALSE	FALSE
TRUE OR FALSE	TRUE
FALSE OR TRUE	TRUE
TRUE OR TRUE	TRUE
FALSE AND FALSE	FALSE
TRUE AND FALSE	FALSE
FALSE AND TRUE	FALSE
TRUE AND TRUE	TRUE
NOT TRUE	FALSE
NOT FALSE	TRUE

(C) copyright 1981 Alcor Systems

D. Operator Precedence

Operator precedence defines the order in which operations take place within expressions. In general, expressions are evaluated from left to right. However, operations of higher precedence are performed before operations of lower precedence. All operators are ranked by precedence. Parentheses have the highest precedence and may be used to alter the normal order of evaluation. Nested parentheses are evaluated from the inside out.

Following is a list of the operators arranged by precedence. Operators listed on the same line have equal precedence.

```
  Highest
Precedence--> ()

              + , -  when used as unary operators

              * , / , DIV , MOD

              + , -

              = , <> , < , > , <= , >= , IN

              NOT

              AND
  Lowest
Precedence--> OR
```

Operation	Equivalent To	Result
8+3*4	8+(3*4)	20
10-8/4*2	10-((8/4)*2)	6
5 MOD 10-5	(5 MOD 10)-5	0
3<2 OR 6>8 AND TRUE	(3<2) OR ((6>8) AND (TRUE))	FALSE
NOT 7*2<5	NOT ((7*2)<5)	TRUE

E. Type Transfer

The type transfer operator is used to temporarily change the type of an existing variable. This is useful when there is a need to reference a variable in a manner which would normally not be allowed by Pascal. For example, you might wish to access the lower and upper byte of an integer variable. The type transfer operator allows you to access parts of variables.

Syntax of type transfer:

```
---> variable ---> :: ---> type id --->
```

A type transferred variable may be used wherever a variable is allowed. Regardless of its original type, the type transferred variable is then accessed according to the type indicated. The type transfer operator tells the compiler to treat the variable as if it were of the new type. No data conversion takes place. The variable is simply referenced as if it were of the new type. Type transferred variables must adhere to the same type matching rules as normal variables.

Example use of type transfer operator:

```
TYPE        byte = 0..#FF;
            integrec = PACKED RECORD
                       upper, lower     :byte;
                       END;
            pointer = @integrec

VAR         number : ARRAY[1..10] OF INTEGER;
            integr : integrec;
            address: pointer;
```

Valid type transfer operations:

```
integr.upper := number[1]::byte;
number[1]::byte := integr.lower;
READ(integr::INTEGER);
number[5] := address::INTEGER;
address::INTEGER := 25 + number[3];
```

The fundamental use of type transfer is to overlay a type template on a data structure so that components of the structure may be treated as if they were of any desired type. This requires a precise understanding of how the compiler represents the data type (how it is stored) in order to insure the operation does what was intended. Because of this, it should be used with caution and only when necessary. (See the System Implementation Manual for information on data representation)

(C) copyright 1981 Alcor Systems

EXPRESSIONS

An expression is a variable, a constant, a function call, a set notation, or a combination of these operands with a description of the operations to be performed on them. The operators and operands of an expression define an implicit type for the expression. When evaluated, the expression yields a value of that type. For example, an integer expression is composed of operands and operators which when evaluated yield an integer result, a real expression yields a value of the type REAL, an ordinal expression yields a value which is of one of the ordinal data types, etc...

An expression can be just a simple expression or it can be a boolean expression. A simple expression can yield a value of any data type. A boolean expression is composed of simple expressions but always yields a value of the type BOOLEAN.

Syntax of expression:

```
  --> boolean expression ---
 |                         |
 |                         V
----> simple expression  ------>
```

Syntax of simple expression:

```
   --> - ---        --- - <--
  |         |      |         |
  |--> + ---|      |--- + <--|
  |         V      V         |
--------------------> term ---->
```

Syntax of term:

```
        --- MOD <--
       |           |
       |--- DIV <--|
       |           |
       |---- /  <--|
       |           |
       |---- *  <--|
       V           |
  -----> factor ----->
```

Syntax of factor:

```
           ---> set notation      ----
          |                           |
          |---> function call     ----|
          |                           |
          |---> variable id       ----|
          |                           |
          |---> constant          ----|
          |                           V
         ----> ( --> expression --> ) ---->
```

For the syntax of set notation, refer to the structured data type SET. A function call has the same form as the procedure statement. The only difference is that a procedure call is a statement while a function call is a part of an expression. Remember that a function has a type associated with it. When a function call is encountered in an expression, the named function is activated. Somewhere in the function a value is assigned to the function name. When the function terminates, the value assigned to it is substituted in the expression for the function call.

Syntax of a function call:

```
                        -----------------------------
                       |                             |
                       |       ------ , <-----       |
                       |      |               |      |
                       |      V               |      V
---> function id ---> ( ---> expression ---> ) --->
```

Example function calls:

```
salary

payment(interestrate,years)

sum(a+b)
```

(C) copyright 1981 Alcor Systems

Example simple expressions:

Expression	Result
time	same type as time
weekday + [saturday,sunday]	set
12*payment(interestrate,years)	integer or real depending on type of function "payment"
entry MOD size	integer
-10 DIV 4 + 9.2/6 -45	real
(var1+var2)*153/(var3-var4)	real

Syntax of boolean expression:

```
      ------ OR <------
     |                 |
     V                 |
 ----> boolean term ---->
```

Syntax of boolean term:

```
      ------ AND <-------
     |                   |
     V                   |
 ----> boolean factor ---->
```

Syntax of boolean factor:

```
   --------    --> relational expression --
  |        |  |                           |
  |        V  |                           V
 ----> NOT -------------> factor --------------->
```

note: factor must be of type BOOLEAN

Syntax of relational expression:

```
                                 --> =  --
                                |         |
                                |--> <  --|
                                |         |
                                |--> >  --|
                                |         |
                                |--> <> --|
                                |         |
                                |--> <= --|
                                |         |
                                |--> >= --|
                                |         V
 ---> simple expression -----> IN -----> simple expression --->
```

Example boolean expressions:

```
a=b OR c<d AND switch

n1 + n2 >= 20 AND n3-n4 <= 11

NOT here OR there

NOT alpha < beta AND gamma <> 'R'

number IN [1..15] OR NOT letter IN ['a'..'z']
```

(C) copyright 1981 Alcor Systems

STATEMENTS

Statements are the Pascal sentences that describe the actions and logic of a program. Statements reside in the statement body part of a block.

A statement may be labeled or unlabeled. A labeled statement is used in conjunction with the GOTO statement. If a statement is labeled, the label must be declared in the LABEL declaration part of the block in which the statement appears.

Syntax of a statement:

```
    -----------------
   |                 |
   |                 V
---> label ---> : ---> unlabeled statement --->
```

Syntax of an unlabeled statement:

```
   --> procedure statement  ---
  |                            |
  |-->    GOTO statement    ---|
  |                            |
  |-->    WITH statement    ---|
  |                            |
  |-->    CASE statement    ---|
  |                            |
  |-->     IF statement     ---|
  |                            |
  |-->   REPEAT statement   ---|
  |                            |
  |-->   WHILE statement    ---|
  |                            |
  |-->    FOR statement     ---|
  |                            |
  |-->  compound statement  ---|
  |                            |
  |--> assignment statement ---|
  |                            V
------------------------------------>
```

(C) copyright 1981 Alcor Systems

A. The Assignment Statement

The assignment statement is used to assign values to variables and function identifiers.

Syntax of the assignment statement:

```
           --> function id ---
          |                   |
          |                   V
        ----> variable id ------> := --> expression -->
```

The action of the assignment statement is to give the variable or function identifier on the left side of the equal sign, the value of the evaluated expression on the right side. The variable may be of any type except FILE. In general, the type of the variable or function must be the same as the type of the evaluated expression. However, there are some exceptions. An identifier of type REAL may be assigned a value which is an integer or a subrange thereof. One side may be a subrange of the other but the value to be assigned should be in the range of the left side. If the identifier on the left side is a SET type, it may be assigned to a set which differs in type as long as the set members of the right side are allowable members of the set on the left side.

note: Variables of type file should not be assigned.

Example assignment statements:

Assignment	left hand side identifier types
a := 10	integer or real
x := 100.5 + 49 + 87/12	real
y := abs(10*z-30.3)	real
test := sample < 10	boolean

(C) copyright 1981 Alcor Systems

B. The Compound Statement

Statements which are bracketed by the two keywords BEGIN and END make up what is termed a compound statement. The compound statement is used in places where more than one statement is required. The compound statement is essential for most of the control structures of Pascal. For example, the FOR statement is a control structure used for executing a statement repeatedly for a specified number of times. The compound statement provides the ability to use this construct for executing a sequence of statements rather than just one.

Syntax of the compound statement:

```
                    ------ ; <----
                    |            |
                    V            |
--> BEGIN ---> statement ---> END -->
```

Example compound statement:

```
BEGIN
  a := b * c;
  d := a/10 + 16.9;
  e := d - 28.3 + 14;
END
```

C. Repetitve Statements

Repetitive statements are the structures used for loop control. They specify that a statement or sequence of statements is to executed repeatedly until some terminating condition occurs. Pascal provides three such control structures.

(C) copyright 1981 Alcor Systems

C.1 The FOR Statement

The FOR loop is used when a statement is to be executed a predefined number of times. The FOR loop is characterized by a loop variable which serves as a counter for controlling the number of times a statement is executed. The counter has defined starting and ending values which are ordinal expressions. The expressions are evaluated once upon entry into the loop. At the beginning of each time through the loop, the counters value is compared to the ending value to determine whether or not to end execution of the FOR. At the end of each time through the loop the counters value changes by 1. If the keyword TO is used, the counter is incremented each time through the loop, while the use of the keyword DOWNTO causes the counter to be decremented. The loop is terminated when the counter has incremented or decremented past the ending value. The FOR statement is not executed if the counters starting value is such that the ending value would never be reached. For example, if the starting value was -1, the ending value was 2, and DOWNTO was used, the FOR statement would not be executed.

Note:
The loop control variable (counter) of a FOR statement need not be declared. If the declaration is absent, the compiler automatically makes the declaration.
If the declaration in not explicit, then the loop control variable becomes undefined upon loop termination.
Also, global variables cannot be used as loop control variables. (for a definition of global, see chapter 9)

Syntax of the FOR statement: (counter must be ordinal type)

```
                                                   --> DOWNTO ---
                                                  |              |
                                                  |              V
--> FOR --> counter id --> := --> ord expr ------> TO ---------
                                                                |
             --------------------------------------------------
            |
             ---> ord expr --> DO --> statement -->
```

Example FOR statements:

```
FOR i := 1 TO 30 DO writeln(' this gets written 30 times')
---------
FOR j := first DOWNTO last DO
  BEGIN
  initialscore[j] := 0;
  time[j] := 60;
  END
```

C.2 The WHILE Statement

The WHILE statement uses a boolean expression to control repeated execution of a statement.

Syntax of the WHILE statement:

```
--> WHILE --> boolean expr --> DO --> statement -->
```

The evaluation of the boolean expression precedes the execution of the statement. If the expression evaluates to TRUE, the statement is executed and then the expression is reevaluated. This loop continues until the expression evaluates to FALSE. The first occurrence of a FALSE evaluation causes termination of the WHILE statement.

Example WHILE statements:

```
WHILE NOT EOLN DO READ(character)

--------

WHILE (a<b) AND (b<c) DO
     BEGIN
     WRITELN(a,b,c);
     a := a + 1;
     c := c - 1;
     END
```

C.3 The REPEAT Statement

The REPEAT statement, like the WHILE, uses a boolean expression to control repeated execution.

Syntax of the REPEAT statement:

```
                ------ ; <----
               |              |
               V              |
--> REPEAT ---> statement ---> UNTIL --> boolean expr -->
```

The REPEAT statement is defined such that a sequence of statements which are bracketed by the two keywords REPEAT and UNTIL will be executed at least once. Following the keyword UNTIL is a boolean expression. If the expression evaluates to FALSE then execution returns to the first statement following the REPEAT keyword. If the expression evaluates to TRUE then execution continues with the statement following the boolean expression.

Example REPEAT statement:

```
REPEAT
  i    := i+1;
  j    := j-1;
  k[j] := (i + j) MOD 100;
  l[i] := (i + j) MOD 200;
UNTIL i=j
```

D. Conditional Statements

Conditional statements are used when the execution of a statement must be controlled by some predetermined condition or when one statement out of a group of statements is to be selected for execution. There are two conditional statements.

D.1 The IF Statement

The IF statement uses a boolean expression to control the execution of statements.

Syntax of the IF statement:

```
                                               ------------------------
                                              |                        |
                                              |                        V
--> IF --> bool expr --> THEN --> statement  ---> ELSE --> statement --->
```

In its simplest form, the IF statement involves the evaluation of a boolean expression to determine whether or not to execute an associated statement which follows the keyword THEN. If the expression is TRUE, then the statement is executed, otherwise it is not. The IF statement can also contain an ELSE clause. In this form, if the boolean expression is TRUE, then the statement following the keyword THEN is executed, otherwise the statement following the keyword ELSE is executed.

Example IF statements:

```
IF finished THEN WRITELN(' operation complete');


IF number < 10 THEN range := 1 ELSE range :=2;


IF alpha >= '0' AND alpha <= '9' THEN digit(alpha)
ELSE
  IF alpha >= 'A' AND alpha <= 'Z' THEN letter(alpha)
  ELSE
    special(alpha);


IF contextlist = NIL THEN
  BEGIN
  NEW(context);
  context@.link := NIL;
  contextlist := context;
  END
ELSE
  BEGIN
  temp := context;
  NEW(context);
  temp@.link := context;
  context@.link := NIL;
  END;
```

The statements following the keywords THEN or ELSE can themselves be IF statements. In some forms, an ambiguity can exist in determining which ELSE clause goes with which IF. For example, consider the following case where b1 and b2 represent boolean expressions and s1 and s2 represent statements.

```
IF b1 THEN IF b2 THEN s1 ELSE s2
```

The ELSE could go with the first IF or the second IF. The rule used for solving the ambiguity is to associate an ELSE clause with the nearest IF. The above statement would then be equivalent to:

```
IF b1 THEN
     BEGIN
     IF b2 THEN s1 ELSE s2
     END
```

Caution: Semicolons must not appear in the middle of a statement. The most common error for beginning programmers is to put a semicolon in an IF statement which has an ELSE clause. While semicolons are necessary for separation of the individual statements within a compound statement, they must not separate an ELSE from its corresponding IF.

D.2 The CASE Statement

The CASE statement uses an ordinal expression to select one statement out of a group of statements for execution. The group of statements represent alternatives. When a CASE statement is executed, one of the alternatives is selected and executed and then control passes to the statement following the CASE statement.

Syntax of the CASE statement:

```
--> CASE --> ord expr --> OF -----------------
                                 |             |
  -------------------------------              |
 |                                             |
 |                                             |
 |   ---------------- ; <----------------      |
 |  |                                    |     |
 |  |  ------ , <---                     |     |
 |  |  |            |                    |     |
 |  V  V            |                    |     V
  ------> constant ---> : --> statement ---->---
                                                 |
  -----------------------------------------------
 |
 |     ----------------------------
 |    |                            |
 |    |                            V
  ------> OTHERWISE --> statement ---> END -->
```

The alternative statements of a CASE statement are preceded by constants. The ordinal expression is evaluated and compared to the constants preceding the alternative statements. If a match is found, the statement which has the preceding constant that matches the evaluated expression is executed. There are two actions which can take place in the event that no match is found. By using the OTHERWISE clause, you may specify a statement to be executed when no match is found. If the OTHERWISE clause is omitted and no match is found, then execution continues with the statement which follows the CASE statement.

Example CASE statements:

```
CASE n1+n2 OF
  10: x := sin(x);
  11: x := cos(x);
  12: x := ln(x);
END;
```

```
CASE ch OF
  'a','b','c': token := 0;
  'd','e','f': token := 1;
  OTHERWISE    token := 2;
END;
```

```
CASE day OF
  monday      : snack := apple;
  tuesday     : snack := orange;
  wednesday   : snack := grapes;
  thursday    : snack := pear;
  friday      : snack := candy;
  saturday,
  sunday      : BEGIN
                weekend := TRUE;
                snack   := nothing;
                END;
END;
```

E. The WITH Statement

The WITH statement is used in conjunction with variables of type RECORD. It makes it possible to use a shorter notation when referencing fields of record variables.

Syntax of the WITH statement:

```
                  ------ , <----
                 |              |
                 V              |
--> WITH ----> variable -----> DO --> statement -->
```

The variable list specifies the record variables whose fields are to be referenced simply by specifiying the field name itself. When fields of a record are nested (ie. a record is defined as a field of another record), the record variable and the fields, down to the level of the field which is to be referenced in short notation, may be specified in the variable list. Then the nested field can be referenced in the statement simply by specifying its field name. There is a conflict inside the WITH statement when an identifier corresponds to both a variable name and a field name of one of the specified records. For example, you could have a record variable named "weekday" with a field named "monday" and also a simple variable named "monday". Then the following WITH statement might be used.

```
WITH weekday DO monday := 1
```

In such a case, the field name takes precedence over the variable name and the field of the record is referenced. If nested WITH statements are used and a field name inside occurs in more than one of the specified records, then the closest WITH takes precedence.

Example WITH statements:

Assume the declarations:

```
customer : RECORD
           name,
           address,
           city      :PACKED ARRAY[1..20] OF CHAR;
           date      :RECORD
                      month,
                      day,
                      year          :INTEGER;
                      END;
           END;
----------------

WITH customer DO
  BEGIN
  name    :=  'JACK SLATE          ';
  address :=  '1216 MELODY LANE    ';
  city    :=  'TULSA, OKLAHOMA     ';
  END;

WITH customer.date DO
  BEGIN
  month   := 10;
  day     := 23;
  year    := 1981;
  END;
```

F. The GOTO Statement

The GOTO statement is used to cause an unconditional branch to a labeled statement.

Syntax of the GOTO statement:

```
--> GOTO --> label -->
```

The label must be declared in the LABEL declaration part of the same block which contains the GOTO referencing it. The GOTO statement cannot specify a branch to a label outside the block in which it resides. Care must be taken when using the GOTO statement. For example, you should not branch inside a FOR loop from a statement outside the loop. This could cause some very unpredictable results.

Example GOTO statement:

```
        FOR i := 1 TO 1000 DO
          IF a(i) <> b(i) THEN GOTO 10
          ELSE a(i) := b(i);
    10: a(i) := ´#0D´;
```

G. The Procedure Statement

The procedure statement causes the activation of a procedure. Control passes to the named procedure and then returns to the statement following the procedure statement when the activated procedure terminates. If a procedure has a parameter list, a procedure statement which activates it must specify an argument for each parameter of the parameter list. The arguments must match the order and type of the parameters specified in the parameter list of the procedure. An argument is specified as an expression. If a parameter of a procedure is a pass by reference parameter (denoted by VAR), the corresponding argument of a procedure statement must be a single variable name. The variable may be a simple variable or a component of a structured variable.

(C) copyright 1981 Alcor Systems

Syntax of a procedure statement:

```
                         -------------------------
                        |                         |
                        |      --- , <--          |
                        |     |         |         |
                        |     V         |         V
--> procedure id ---> ( ---> expr --->  ) --->
```

Example procedure statement (call):

(See the procedure declaration in section B.7 of chapter 2)

```
getvalue(n+j,8,hexstring,value)

report

writeout(x,y,3.7+9.6/z)
```

(C) copyright 1981 Alcor Systems

PROCEDURES AND FUNCTIONS

(See chapter 2 for a description of the syntax of procedure and function declarations. A discussion of parameter passing is included with the discussion of the procedure heading.)

Procedures and functions are the tools used to modularize a program. This is the process of breaking a program up into smaller and more manageable pieces. They make a program much more readable and make possible later modifications much easier to handle.

Procedures and functions can be compiled separately and then linked to programs that use them. This allows for the development of libraries of commonly used procedures and functions. Then all the programs that use them can link them in rather than having to include them in the program itself.

The variables declared in a procedure or function do not occupy storage space until the procedure or function is activated. When activated, storage space is allocated for the variables and when the procedure or function terminates, the allocated space is released. Therefore, the amount of storage (or stack) space required by a program at any point in time is a function of the number of blocks which are activated at that time.

A procedure is activated (or called) by a procedure statement. When a procedure is called, control is passed from the point of the call to the procedure. The statements in the procedure then are executed. When the block END of the procedure is reached or when a call to the ESCAPE procedure is made, control passes back to the statement following that which activated the procedure.

A function is activated by an expression. When an expression which contains a reference to a function is evaluated, the function reference causes control to pass to the named function. The statements in the function then are executed. Unlike procedures, functions have a declared type. At some point inside the statement body of a function, the function name should be assigned a value. The value must be the same type as the type to which the function is declared. When the block END of the function is reached or when a call to the ESCAPE procedure is made, control passes back to the evaluation of the expression which activated the function and the function reference is replaced by the value assigned to the function.

(C) copyright 1981 Alcor Systems

A. Scope Rules

A procedure or function declaration forms a new block which is a subblock of the block in which the declaration appears. The new block formed is "nested" within the block which declares it. This process of nesting which occurs every time a procedure or function is declared produces a program structure such as the one shown on the first page of chapter 2. Any block which is enclosed by another block is said to be nested within that block. The level numbers on the diagram indicate how deep the nesting goes beyond the program block which is arbitrarily assigned level 1. The existence of procedures and functions makes it necessary to talk about scope rules. Scope rules describe the accessibility of identifiers from any particular place in a program. The two terms local and global are helpful in discussing scope rules.

An identifier is considered to be local to a block if the identifier is declared within the same block. If there are no blocks nested within the declaring block, then a local identifier can only be referenced by the block which declares it. Enclosing blocks cannot access a local identifier.

An identifier is considered to be global to blocks which are nested within the block in which the identifier is declared. If an identifier is global to a particular block, then that block can reference the identifier provided that it has not declared an identifier of the same name. If a block declares an identifier with the same name as a global identifier, then the global identifier is no longer accessible from that block. Also, any further nested blocks will not have access to the original global identifier.

Identifiers declared in the program block are accessible from any place in a program because all other blocks are nested within the program block. Therefore, identifiers declared in the program block are global to all procedures and functions of the program. Identifiers declared in a procedure or function are local to that procedure or function. The only places in the program which can access these identifiers are the procedure or function itself and the procedures or functions, if any, which are nested within. The nested procedures or functions can access only the global identifiers which they do not declare themselves.

A procedure or function declaration consists of a heading followed by a block. It is important to note that the procedure or function name of a heading is local to the block which declares it. The parameters of the heading are local to the procedure or function itself. This means for example that a procedure statement in the program block can reference any procedure declared in the program block. However, a procedure statement in the program block can not reference any procedure declared within one these procedures.

As an example of how scoping effects the accessibility of identifiers, consider the sample diagram on the first page of chapter 2. The following table shows for each block of the diagram, the procedures and functions which are callable from that block, and the constants, types, variables, etc. which can be referenced by the block.

Block	accessible procedures and functions	accessible constants, types, variables, etc.
A	B, D, F	A
B	B, C, D, F	A, B
C	B, C, D, F	A, B, C
D	B, D, E, F	A, D
E	B, D, E, F	A, D, E
F	B, D, F	A, F

(C) copyright 1981 Alcor Systems

B. FORWARD

The rule that an identifier must be declared before it is referenced means that a procedure or function must be declared before it is referenced by a procedure statement or by an expression with a function reference. Some calling sequences that occur among a group of procedures or functions make it impossible to obey this rule. For example, if two procedures call each other, then you can not declare one without referencing the other. The keyword FORWARD provides the mechanism for getting around this problem. Using the keyword FORWARD with just the heading for a procedure or function declaration signals the compiler that the procedure or function block will be declared at some later point in the program. If the procedure or function has parameters, the parameters are declared as well. Then the procedure or function which has been forward declared may be referenced.

Syntax of forward declaring a procedure or function:

```
         --> function heading ---
        |                        |
        |                        V
      ----> procedure heading ----> FORWARD --> ; -->
```

(See chapter 2 for the syntax of procedure and function headings)

The actual declaration of a forward declared procedure or function can appear at some later place in the program. The place that it appears must be at the same level and scope as its forward declaration. The actual declaration consists of the heading with no parameters, followed by the block. Since the parameters were declared in the forward declaration, they must not be declared again in the actual declaration.

If a forward declared procedure or function does not have its actual declaration present, then it is treated as an external procedure or function.

Example use of forward:

```
PROCEDURE abc(pl, p2 : INTEGER); FORWARD;

PROCEDURE xyz;
VAR    pl, p2 : INTEGER;
BEGIN
abc(pl,p2);
END;

PROCEDURE abc;
BEGIN
 .....

END;
```

(C) copyright 1981 Alcor Systems

C. EXTERNAL

An external procedure or function can be declared in a program by specifying its heading followed by the keyword EXTERNAL.

Syntax of externally declared procedures or functions:

```
                                         note: EXTERN also accepted
          --> function heading ---
         |                        |
         |                        V
       -----> procedure heading -----> EXTERNAL --> ; -->
```

Note:
for brevity the word "routine" will be used in place of "procedure or function" in the following discussion.

The linking loader may be used to link separately compiled routines to a program. By declaring a routine to be external, the actual declaration does not have to appear in the program. This is very useful when working with large programs. A large program may be broken up into many routines which are declared as external. The external routines can then be compiled individually. The linking loader can then be used to link the compiled program to its individually compiled routines. One advantage to this is that any changes which are made to a particular routine will cause only that routine to have to be recompiled. The linking process is then repeated after the changed routine has been recompiled. Another advantage is that slightly larger programs can be created by compiling them in pieces and then linking the pieces together.

Perhaps one of most frequent use of external routines is to create a file or library of commonly used routines. Then all the programs which use the routines can link to them rather than having to declare them in each program.

A compiler option must be used to compile a routine by itself. The reason is that a routine by itself is not a legal Pascal program. Therefore, a legal program must be constructed around the routine. This would include a program heading, the environment of the routine, the procedure or function declaration, and a statement body. The environment consists of any constants or types which are in the scope of and are used by the external routine. If global constants or types are needed by the routine, they should be given the same names as those used in the programs that use the routine. The scope refers to the identifiers in a program which are accessible to the externally declared routine.

(C) copyright 1981 Alcor Systems

Variables can also be included in the environment but this is not recommended. If an external routine needs to access a global variable, the variable should be passed as a parameter to the routine. Otherwise, extreme care must be taken to assure that the environment around the external routine matches the environment of the programs which use the routine. The statement body contains the compiler option which is called "nullbody". The nullbody option tells the compiler not to generate any code for the program. Only code for the declared routine is generated.

The syntax for using the nullbody compiler option is shown in the appendix along with all the other compiler options.

Example use of external procedure:

```
PROGRAM sample;
CONST    .....
TYPE     .....
VAR      xmin,xmax,ymin,ymax : REAL;
         .....

PROCEDURE axes(xmin,xmax,ymin,ymax : REAL); EXTERNAL;
BEGIN
    ....
  axes(xmin,xmax,ymin,ymax);
    ....
END.       (*sample*)

----------------------------
```

Separate compile of procedure axes:

```
PROGRAM axesroutine;
(*global environment, if any, goes here*)
PROCEDURE axes(xmin,xmax,ymin,ymax : REAL);
  TYPE  ....
  VAR   ....
  BEGIN
    .....
    .....
  END;      (*procedure axes*)
BEGIN
  (*$NULLBODY*)
END.
```

D. Recursion

Pascal is a language which supports recursion. Recursion refers to having more than one activation of a particular procedure or function at the same time. There are two forms of recursion. Direct recursion refers to a procedure or function which calls itself. Indirect recursion refers to a procedure or function which makes a call which eventually results in the procedure or function being activated again. An example of this is two procedures that call each other. When writing recursive procedures, some conditional statement must exist in the procedure to halt the recursion at some point. Otherwise, there would be an endless loop which would terminate only after the stack was exhausted, crashing the program. Recall that each activation of a procedure results in space being allocated for its local variables.

Example use of recursion:

```
PROCEDURE xyz;
  (*declarations here*)
BEGIN
   ...
   xyz;   (*procedure calls itself*)
   ...
END;
```

(C) copyright 1981 Alcor Systems

E. Predeclared Procedures and Functions

The predeclared procedures and functions are accessible from any place in a program. They are declared in an imaginary block which surrounds the program block. The names of predeclared procedures or functions may be used as identifiers in programs. This means that the name of a predeclared procedure or function may be used in a declaration. If so, then the predeclared procedure or function whose name is used in a declaration is no longer accessible to the program. Its name is associated with the new declaration.

File Associated Procedures

RESET(f)	Positions the file pointer of the specified file to the beginning for the purpose of reading. If the file is empty, then the function EOF becomes true, else it is false.
REWRITE(f)	Replaces the specified file with an empty file. The file pointer is positioned to the beginning of the file.
PAGE(f)	Outputs a formfeed to the specified file. Formfeeds cause skipping to the top of the next page when the file is printed.
CLOSE(f)	Closes the specified file. This procedure should be used if the file variable is not a simple variable to insure the integrity of the file.
MESSAGE(s)	Outputs the specified string to the terminal
READ , READLN	Read data from a device
WRITE, WRITELN	Write data to a device (See chapter 10 for details)

(C) copyright 1981 Alcor Systems

Arithmetic Functions

	Operation	Type of x	Type of Result
ABS(x)	absolute value	integer, real	same type as x
SQR(x)	square	integer, real	same type as x
SIN(x)	sine	integer, real	real
COS(x)	cosine	integer, real	real
ARCTAN(x)	arctangent	integer, real	real
EXP(x)	natural (base e) exponential	integer, real	real
LN(x)	natural logarithm	integer, real	real
SQRT(x)	square root	integer, real	real

Boolean Functions

ODD(x)
Operation: Returns true if x is odd, else false
Type of x: integer
Type of result: boolean

EOLN(x)
Operation: Returns true if the end of a line in the file has been reached
Type of x: text
Type of result: boolean

EOF(x)
Operation: Returns true if the end of the file has been reached.
Type of x: file
Type of result: boolean

(C) copyright 1981 Alcor Systems

Transfer functions

```
TRUNC(x)        Operation: Truncates a real value to its
                           integer part
                Type of x: real
                Type of result: integer

ROUND(x)        Operation: Rounds a real value to the
                           nearest integer
                Type of x: real
                Type of result: integer

ORD(x)          Operation: Returns the ordinal number of x.
                Type of x: any ordinal type
                Type of result: integer

CHR(x)          Operation: Returns the character whose ordinal
                           number is x
                Type of x: integer
                Type of result: char

LOCATION(x)     Operation: Returns the address of variable x
                Type of x: any type
                Type of result: integer

SIZE(x)         Operation: Returns the size of type x in bytes
                Type of x: any type identifier
                Type of result: integer

HB(x)           Operation: Returns the high byte of x
                Type of x: integer
                Type of result: integer

LB(x)           Operation: Returns the low byte of x
                Type of x: integer
                Type of result: integer
```

Data transfer procedures

```
PACK(a,i,z)     Operation: Copy the unpacked array a into the
                           packed array z. If the dimension of a
                           is m..n and the dimension of z is u..v
                           and n-m > v-u then the operation is
                           equivalent to:
                           for j:= u to v do z[j] := a[j-u+i]

UNPACK(z,a,i) Unpacks the above array.
```

(C) copyright 1981 Alcor Systems

Dynamic allocation procedures

NEW(p) Allocates a new variable v and assigns the pointer reference of v to the pointer variable p. Tag field values may appear as parameters to NEW but are non-functional.

DISPOSE(p) Releases the storage occupied by the variable pointed to by p.

Other functions

SUCC(x) Operation: Returns the successor of x which is next higher value in the enumeration of which x is a member
Type of x: any ordinal type
Type of result: same type as x

PRED(x) Operation: Returns the predecessor of x which is the next lower value in the enumeration of which x is a member
Type of x: any ordinal type
Type of result: same type as x

Other procedures

ESCAPE Causes termination of a block just as if the block end had been reached. If the block is a procedure or function, then control returns to the calling block. If the block is the program block, then program execution is terminated.

note : IF files are declared locally within a procedure, then the files must be closed using the procedure CLOSE before calling ESCAPE. Normal termination of a block results in files automatically being closed.

(C) copyright 1981 Alcor Systems

INPUT AND OUTPUT

Input and output is the communication of a program to the external environment. A program communicates to the external environment through the use of logical files. Logical files are the variables in a program which are declared as type FILE or TEXT. The logical files are then associated with physical files. Physical files are the actual devices of the computer system. A physical file could be a disk file, a terminal , a printer, or some other device. The method of associating logical files to physical files is discussed in the System Implementaion Manual.

Predeclared procedures and functions are provided for handling input and output. These procedures and functions have a characteristic unlike other procedures and functions. The number of parameters passed to them can vary. They may be called with no parameters or with several parameters. Since each input and output routine performs an operation on a file, it must know which file to operate on. If a routine is passed the logical file name, then it operates on the specified file, otherwise it operates on a default logical file. The two predeclared variables INPUT and OUTPUT are the default logical files. They are both declared as type TEXT. The one used as the default depends on the routine called. The input routines default to INPUT and the output routines default to OUTPUT.

I/O Routines

Procedures			Functions
input	output	general	
RESET READ READLN	REWRITE WRITE WRITELN PAGE MESSAGE	CLOSE	EOF EOLN

A file has associated with it a file pointer. The file pointer is used to point to an individual component of a file. There are two predeclared boolean functions which may be used to check the status of a files pointer. Both functions may or may not take a logical file name as a parameter. If no file parameter is passed, the default is INPUT. The function EOF(file) returns the value TRUE if the pointer is at the end of the file. Otherwise, the value returned is FALSE. The function EOLN(file) can only be used with files of type TEXT. It returns the value TRUE when the files pointer is at the end of a line. Otherwise, the value returned is FALSE.

Syntax of function EOF or EOLN: (default: file = INPUT)

```
        --> EOLN ---     ---------------------
       |           |   |                     |
       |           V   |                     V
  -----> EOF-----------> ( --> file --> ) --->
```

Examples of using EOF and EOLN:

```
WHILE NOT EOF(datain) DO              IF EOF THEN quit
  BEGIN                               ELSE
  WHILE NOT EOLN(datain) DO             READ(number);
    BEGIN
    READ(ch);
    .....
    END;
  .....
  END;
```

A. RESET

The RESET procedure opens a file so that it can be read from. No input can be received from a file without this operation first being performed on it.

Syntax of RESET: (default: file = INPUT)

```
                ---------------------
               |                     |
               |                     V
  --> RESET ---> ( --> file --> ) --->
```

The procedure positions the file pointer to the beginning of the file. If the file is empty, then the function EOF(file) becomes TRUE. If the file is not empty, then the function EOF(file) becomes FALSE. The statement RESET(INPUT) is implicitly executed at the beginning of a program that contains either implicit or explicit reference to the logical file INPUT. Therefore, it is not necessary for a program to open the default logical file INPUT.

Example use of RESET:

```
PROGRAM readdata;
VAR    datain : TEXT;
BEGIN
  RESET(datain);      (*open file datain for reading*)
  .....
END.
```

Input and output to files is buffered. This is to prevent having to access a physical device every time an operation is performed. Each file used by a program has an associated buffer. Unlike standard Pascal, the input buffer of a file is not filled when a reset is performed. The input buffer becomes filled the first time a READ, READLN, EOLN, or EOF is performed on the file. This prevents the normal problems associated with reading from a terminal. Programs can have their logical files remapped from a disk file to a terminal without modification to the program itself.

(See the System Implementation Manual for a description of how to associate logical files to physical files)

B. REWRITE

The REWRITE procedure opens a file so that it can be written to. No output can be sent to a file without this operation first being performed on it.

Syntax of REWRITE: (default: file = OUTPUT)

```
                  ---------------------
                 |                     |
                 |                     V
--> REWRITE ---> ( --> file --> ) --->
```

The procedure positions the file pointer to the beginning of the file. The file becomes empty when this happens. This means that any data in the file is lost.

The statement REWRITE(OUTPUT) is implicitily executed at the beginning of a program that contains either implicit or explicit reference to the logical file OUTPUT. Therefore, it is not necessary for a program to open the default logical file OUTPUT.

C. READ

The READ procedure assigns the value of components of a file to variables.

Syntax of READ: (default: file = INPUT)

```
                  ---------------     ----- , <---
                 |               |   |            |
                 |               V   V            |
--> READ --> ( ---> file --> , -------> variable ---> ) -->
```

The number of variables passed to the procedure determines the number of components read from the file. The components refer to the way the file is logically separated into individual data elements. Each component is of some data type which defines its size. Reading begins with the component pointed to by the file pointer. The first variable specified is assigned the value of this component and then the file pointer is advanced to the next component. This process is continued until all the variables specified are assigned values. The type of each variable must match the type of the file component being assigned to it.

Text files

If the file is of type TEXT, the variables can be type REAL, INTEGER, subrange of integer, CHAR, or strings. Strings are declared as single dimensioned packed arrays of the type CHAR. These types can be intermixed as components of text files. Then they may be read by specifiying variables which match in type and order, the components of the file.

(C) copyright 1981 Alcor Systems

If the variable is of type CHAR, then a single character is read from the file. If the variable is an array of CHAR, then the dimension of the array determines the number of characters read from the file. If an end of line or file mark is encountered before the array is full, then the characters read up to that point are left justified in the array and the remaining elements are filled with blanks. Integer and real numbers are represented in files as strings of characters. Individual numbers in a file are separated by blanks or by an end of line mark. When a number is read, the character string representing the number is automatically converted to its real or integer value before being assigned to the variable. With text files, consecutive read operations automatically skip end of line marks when reading integer, real, or boolean variables. When reading character or string variables, the end of line mark is not skipped. In this case, the procedure READLN must be executed to cause the file pointer to advance to the next line.

Example use with text files:

Consider the following file of data:

```
SAM JONES     25     183.5     369
MARY SMITH    23     105.4     356
       ......
       ......
```

and the declarations:

```
VAR      name           : PACKED ARRAY[1..10] OF CHAR;
         number, total  : INTEGER;
         score          : REAL;
         students       : TEXT;
```

If the file pointer of "students" points to the beginning of a line (it does immediately after a RESET) then:

```
READ(students,name,number,score,total)
```

would assign a string, integer, real, and integer value to the 4 specified variables. The file pointer would then point to the character immediately following the last value read.

(C) copyright 1981 Alcor Systems

Non-text files

If the file is not of type TEXT, then all components of the file are of the same type. The components of a file may be declared to be of any type except the type FILE or structured types containing a component of type FILE. This means for example, that you could declare a file of records. Then an entire record can be read into a variable of the same record type. This however, requires that the file of records has previously been created through the use of the procedure WRITE. The reason for this is that all files which are not of type TEXT are read and written in binary form.

Example use with non-text files:

assume the following declarations:

```
TYPE      food = RECORD
                 fruit     : (orange, grape, apple);
                 vegetable : (corn, okra, beans);
                 cost      : INTEGER;
                 END;

VAR       groceries : FILE OF food;
          item      : food;
```

---

then:

```
          READ(groceries, item)
```

would assign one record from the file to the variable "item".

Care should be taken not to read past the end of a file. The function EOF is provided for preventing this from occuring. The program will not abort if you try to read past the end of file, but the value assigned to the variable will be some unknown value.

D. WRITE

The procedure WRITE appends values to a file. The number of values passed to the procedure determines the number of values output to the file. If a file is declared as type TEXT, then output values can be specified as strings or expressions. If a file is declared as a type other than type TEXT, then the output values are restricted to variables of the same type only.

Syntax of WRITE:

For non-text files:

```
                                        ---    , <----
                                        |             |
                                        V             |
--> WRITE --> ( --> file --> , ---> variable ---> ) -->
```

For text files: (default: file = OUTPUT)

```
                 ---------------   -------- , <--------
                 |              | |                    |
                 |              V V                    |
--> WRITE --> ( ---> file --> , -----> write parameter ---> ) -->
```

Syntax of write parameter:

```
                     ---------------------------
                     |                          |
 -->  real expr    --> : --> integer expr ------> : --> integer expr --
|                                               |                      |
|--> integer expr ---                           V                      |
|                    |  ------------------------|                      |
|--> boolean expr ---| |                        |                      |
|                    V |                        V                      V
---->   string      -------> : --> integer expr ----------------------->
```

Syntax of string: (string variable = packed array of char)

```
        ---- string variable -------
       |                            |
       |       ------ , <----       |
       |       |             |      |
       |       V             |      V
     ---> ' ---> character ---> ' --->
```

Text files

If the file is of type TEXT, then the values output to the file may be specified as strings or as boolean, integer, or real expressions. If a string is specified, then the characters of the string are output to the file. If a boolean expression is specified, then either the characters 'TRUE ' or 'FALSE' are output to the file depending on the value of the expression. If an integer or real expression is specified, then the value of the expression is converted to a character string before being output to the file. An integer expression may be output in hexadecimal or decimal base representation.

The number of characters to output for a value can be specified by an integer expression which follows the value, separated by ":". If the number of characters is not specified for a particular value, then a default number of characters will be output.

---For a string---

If the number is less than the length of the string, then all the characters of the string are output. If the number is greater than the length of the string, then blanks will be appended to the string. The default number is the length of the string.

Example: `WRITE(' literal string' : 20)`

---For a boolean expression---

The same rule applies for the strings 'FALSE' and 'TRUE'.

Example: `WRITE(a AND b)`

---For an integer expression---

If the number is less than the number of digits in the integer, then all the digits are output. If the number is greater than the number of digits, then the excess characters are output as blanks before the integer is output. The default number of digits for integers is 8. An integer value may be written in hexidecimal base format by specifying : width HEX

Example: `WRITE(outfile, n+5 :i)`

---For a real expression---

Two numbers may be specified for real values. The first number specifies the total number of digits for the mantissa. The second specifies the number of digits after the decimal point. For a description of the effects of these specifications and for the defaults used, see the System Implementation Manual.

Example: `WRITE(2.5*random :5:3, random/x:3:6)`

Non-text files

If the file is not of type TEXT, then output values must be variables. Output directed to non-text files is in binary form. This means that values are output in the same form as they are stored. For example, an integer is not converted to a character string before it is output.

Example use with non-text files:

```
WRITE(groceries,item)
```

E. READLN

This procedure can be used only with files of type TEXT. (See section C.1 of chapter 4 for a description of text files.)

The READLN procedure is similar to the READ procedure. The difference is that at the end of the read operation, the file pointer is advanced to the beginning of the next line.

Syntax of READLN: (default: file = INPUT)

```
            -----------------------------------------------
           |                                               |
           |      ---------------    ----- , <----         |
           |     |               |  |             |        |
           |     |               V  V             |        V
--> READLN ---> ( ---> file ---> , -----> variable -----> ) --->
                          |                          ^
                          |                          |
                           --------------------------
```

The READLN procedure may be called without passing any variables to be read. When no variables are specified, then the procedure just advances the line pointer to the beginning of the next line.

The statement: READLN(var1,var2,var3)
is equivalent to: BEGIN READ(var1,var2,var3); READLN END

The function EOLN can be used to determine whether or not a files pointer is at the end of a line.

Example use of READLN:

```
i := 0;
WHILE NOT EOF DO
  BEGIN
  i := i+1;
  READLN(a[i]) (*reads one value from each line*)
  ......
  END;
```

```
WHILE NOT EOF (infile) DO
  BEGIN
  WHILE NOT EOLN(infile) DO
    BEGIN
    READ(infile,ch);
    .....
    END;
  READLN(infile);  (*advances file pointer to next line*)
  END;
```

F. WRITELN

This procedure can only be used with files of type TEXT. (See section C.1 of chapter 4 for a description of text files).
The WRITELN procedure is similar to the WRITE procedure. The difference is that at the end of the write operation, an end of line mark is appended to the file.

Syntax of WRITELN: (default: file = OUTPUT)

```
            ----------------------------------------------------
           |                                                    |
           |     ----------------   --------- , <--------       |
           |    |                |  |                    |      |
           |    |                V  V                    |      V
--> WRITELN ---> ( ---> file ---> , -----> write parameter ---> ) --->
                             |                        ^
                             |                        |
                              ------------------------
```

(See WRITE for syntax of write parameter)

(C) copyright 1981 Alcor Systems

The WRITELN procedure may be called without passing any values to written. When no values are specified, then the procedure just appends an end of line mark to the file.

```
The statement:        WRITELN(var1,var2,var3)
is equivalent to:     BEGIN WRITE(var1,var2,var3); WRITELN END
```

Example use of WRITELN:

```
(*writes 2 values on each line*)
FOR k := 1 TO 100 DO WRITELN(a[k],b[k]);
```

```
FOR j := 1 TO maximum DO
  BEGIN
  i := 0;
  REPEAT
    i := i+1;
    WRITE(number[j]);
  UNTIL (i=5) OR (number[j]>100);
  WRITELN;              (*advance file pointer to next line*)
  END;
```

G. CLOSE

The use of the CLOSE procedure will assure that file data will not be lost if the program abnormally terminates and does not properly close the file. It may also be used in conjunction with the external runtime routine SET$ACNM.
(see System Implementation manual)

Syntax of CLOSE:

```
--> CLOSE --> ( --> file --> ) -->
```

H. PAGE

The PAGE procedure appends a formfeed to a file. Formfeeds cause printers to skip to the top of the next page. This procedure provides a way of controlling the number of lines printed on a page.

This procedure may only be used with files of type TEXT.

Syntax of PAGE: (default: file = OUTPUT)

```
                  ---------------------
                  |                   |
                  |                   V
  --> PAGE ---> ( --> file --> ) --->
```

I. MESSAGE

The procedure MESSAGE may be used to output strings to the terminal. It takes one parameter which is either a string constant or variable. A string constant is a sequence of characters enclosed in single quotes. A string variable is a variable declared as a packed array of characters.

Syntax of MESSAGE:

```
  --> MESSAGE --> ( --> string --> ) -->
```

Programs which require only string output to the terminal can use this procedure rather than the WRITE procedure.

Example use of MESSAGE:

```
MESSAGE(' time to quit');

MESSAGE(string);
```

(C) copyright 1981 Alcor Systems

APPENDIX

A. COMPILER OPTIONS

Compiler options are provided to change the behavior of the Pascal compiler. These options allow features to be enabled or disabled and can alter the code generated at compile time.

Compiler options are specified in comments. A comment that contains a dollar sign as the first character specifies an option. All compiler options have two states, on and off. An option is turned on by placing its name after the dollar sign. If the option name is preceded by the word "NO", then the option is turned off. For example, the following line will cause all real variables to be double precision.

```
(*$DOUBLE*)
```

DOUBLE

This option specifies that all real variables within the program should be double precision. This option must precede the program statement. If it occurs anywhere else in the program, it will be ignored. If the option is off (the default), then real variables are single precision.

Example:

```
(*$DOUBLE*)
PROGRAM DBL;
VAR
   R : REAL;
BEGIN
END.
```

In this program, the variable "R" will be declared as double precision.

(C) copyright 1981 Alcor Systems

FORDECL

This option is used to change the behavior of loop counters in FOR statements. If the option is turned on(default is off), then all FOR loop counters are treated as temporary variables. They do not need to be declared, and even if a declaration is present, a new variable is used rather than the declared variable. These FOR loop counters are defined only within the loop and disappear when the loop is exited.

> note: FOR loop counters are automatically declared if not explicitly declared even with this option off. However, temporary variables are not used in place of declared variables.

Example:

```
PROGRAM FORLOOP;
(*$FORDECL*)
VAR
   I : INTEGER;
BEGIN
   A := 0;
   I := 0;
   FOR I := 0 TO 4 DO A := A + I;
   WRITELN(OUTPUT,I,A);
END.
```

In the above program, the I used as a FOR loop counter is a different variable from the I declared in the VAR section. When the write statement is executed, the values 0 and 10 will be printed.

INOUT

This option enables the predeclared files: INPUT and OUTPUT(default is on). If this option is turned off before the PROGRAM statement, then the files input and output will not be declared. This option prevents the reset of INPUT and the rewrite of OUTPUT and can be used in programs that do not perform normal pascal input and output.

IF

The if option provides conditional compilation. The word IF is followed by the name of a boolean constant. If the constant has the value "TRUE", then compilation continues as if the option had not been present. If the constant has the value "FALSE" then compilation stops at that point, and all text is treated as comments until a (*$NO IF*) is encountered. Note that IF options do not nest. That is, an IF option should not occur within the scope of another if option. The if option can be used to configure a program for different environments with minimum changes to the source. It is also useful for removing debugging statements once the program is working properly.

Example:

```
PROGRAM Test;
CONST
   debug = false;

FUNCTION FACTORIAL(I : INTEGER) : REAL;
BEGIN
   IF I = 0 THEN FACTORIAL := 1
   ELSE BEGIN
      (*$IF DEBUG*)
      WRITELN(OUTPUT,'CALLING FACTORIAL(',I-1,')');
      (*$NO IF*)
      FACTORIAL := I * FACTORIAL(I-1);
      END;
END; (* FACTORIAL *)

BEGIN
   WRITELN(OUTPUT,'FACTORIAL(20)=',FACTORIAL(20));
END.
```

In the above program, the write statement within the recursive function FACTORIAL could be turned on during debugging by setting debug to TRUE. Once the program is running, it can be recompiled with debug set to FALSE. The write statement will be effectively removed. In fact, since no code is generated for it, the resulting object program will be shorter. This has the same effect as removing the statement with an editor or placing open and close comments arround it. The advantage is that many statements can be disabled or enabled with a single change to the source program. Also, it is simple to reenable debugging statements should it become necessary in the future.

NULLBODY

The nullbody option is used to disable code generation for a procedure, function or program. The nullbody option should occur after the BEGIN that starts the block and before any executable statements. Nullbody will prevent code from being generated and can be used when procedures are being compiled separately. Since every program must have a program statement and a main program body, it is necessary to use nullbody to disable code generation for the main program when a subroutine library is being compiled.

For example:

```
PROGRAM SUBLIBRARY;
TYPE
   STRING = PACKED ARRAY[1..80] OF CHAR;

PROCEDURE CONCATENATE(VAR S1, S2, RESULT: STRING);
BEGIN
 (* BODY OF CONCATENATE *)
END;

PROCEDURE MID$(VAR S : STRING; FIRST, LAST : INTEGER;
              VAR RESULT : STRING);
BEGIN
 (* BODY OF MID$ *)
END;

BEGIN
  (*$NULLBODY*)
END.
```

If the above program is compiled, the object file will contain code only for the two procedures: CONCATENATE and MID$. There will be no main program. This allows these procedure to be linked to another program.

B. ERROR MESSAGES

2 IDENTIFIER EXPECTED
3 'PROGRAM' EXPECTED
4 ')' EXPECTED
5 ':' EXPECTED
6 ILLEGAL SYMBOL
8 'OF' EXPECTED
9 '(' EXPECTED
10 ERROR IN TYPE
11 LEFT BRACKET '[' OR '(.' EXPECTED
12 RIGHT BRACKET ']' OR '.)' EXPECTED
13 'END' EXPECTED
14 ';' EXPECTED
15 INTEGER EXPECTED
16 '=' EXPECTED
17 'BEGIN' EXPECTED
20 ',' EXPECTED
22 '..' EXPECTED
23 '.' EXPECTED
49 'ARRAY' EXPECTED
50 CONSTANT EXPECTED
51 ':=' EXPECTED
52 'THEN' EXPECTED
53 'UNTIL' EXPECTED
54 'DO' EXPECTED
55 'TO'/'DOWNTO' EXPECTED
57 'FILE' EXPECTED
66 TYPE IDENTIFIER EXPECTED
80 OPEN COMMENT WITHIN A COMMENT
81 UNKNOWN OPTION
82 # REQUIRES A 2 CHARACTER HEX VALUE OR ##
101 IDENTIFIER DECLARED TWICE
102 LOWER BOUND EXCEEDS UPPER BOUND
103 IDENTIFIER IS NOT OF APPROPRIATE CLASS
104 UNDECLARED IDENTIFIER
105 CLASS OF IDENTIFER IS NOT VARIABLE
107 INCOMPATIBLE SUBRANGE TYPES
113 ARRAY BOUNDS MUST BE SCALAR
117 UNSATISFIED FORWARD REFERENCE TO A TYPE IDENTIFER OF A POINTER
119 ';' EXPECTED (PARAMETER LIST NOT ALLOWED)
120 FUNCTION RESULT MUST BE SCALAR, SUBRANGE, OR POINTER
123 FUNCTION RESULT EXPECTED
126 IMPROPER NUMBER OF PARAMETERS
127 TYPE OF ACTUAL PARAMETER DOES NOT MATCH FORMAL PARAMETER
129 TYPE CONFLICT OF OPERANDS IN AN EXPRESSION
132 COMPARISON WITH '>' OR '<' NOT ALLOWED ON SETS
134 ILLEGAL TYPE OF OPERANDS
135 TYPE OF EXPRESSION MUST BE BOOLEAN

136 SET ELEMENT TYPE MUST BE SOME ENUMERATION TYPE
138 TYPE OF VARIABLE IS NOT ARRAY
140 TYPE OF VARIABLE IS NOT RECORD
141 TYPE OF VARIABLE IS NOT POINTER
148 SET BOUNDS OUT OF RANGE
152 NO SUCH FIELD IN THIS RECORD
154 ACTUAL PARAMETER MUST BE A VARIABLE
156 MULTIDEFINED CASE LABEL
161 PROCEDURE OR FUNCTION ALREADY DECLARED AT A PREVIOUS LEVEL
165 LABEL ALREADY DEFINED
167 UNDECLARED LABEL
168 LABEL NOT DEFINED
182 "FOR" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
183 "CASE" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
184 "FOR" VARIABLE MUST BE LOCAL
185 OPERATION DEFINED FOR TEXT ONLY
186 OPERATION NOT DEFINED FOR TEXT FILES
193 ACCESS STATEMENT MISSING FOR COMMON
199 FEATURE NOT IMPLEMENTED
202 STRING CONSTANT CANNOT SPAN LINES
203 INTEGER CONSTANT TOO LARGE
210 FIELD WIDTH MUST BE INTEGER
211 FRACTION LENGTH MUST BE OF TYPE INTEGER
212 HEX FORMAT ALLOWED ONLY FOR TYPE INTEGER
219 PARAMETER MUST BE OF TYPE FILE
220 PARAMETER MUST BE OF TYPE INTEGER
223 PARAMETER MUST BE OF TYPE POINTER
230 ILLEGAL TYPE OF PARAMETER IN STANDARD PROCEDURE CALL
250 TOO MANY NESTED SCOPES - LIMIT IS 15
401 OPEN COMMENT ENCOUNTERED IN A COMMENT
403 TO MANY PROCEDURE NESTING LEVELS
404 ARRAY BOUNDS MUST BE SCALAR

C. Standard 7-bit USASCII Character Set

Decimal	Octal	Hex	Graphic	Name
0.	000	00	^@	NUL (used for padding) <null>
1.	001	01	^A	SOH (start of header)
2.	002	02	^B	STX (start of text)
3.	003	03	^C	ETX (end of text)
4.	004	04	^D	EOT (end of transmission)
5.	005	05	^E	ENQ (enquiry)
6.	006	06	^F	ACK (acknowledge)
7.	007	07	^G	BEL (bell or alarm)
8.	010	08	^H	BS (backspace) <bs>
9.	011	09	^I	HT (horizontal tab) <tab>
10.	012	0A	^J	LF (line feed) <lf>
11.	013	0B	^K	VT (vertical tab)
12.	014	0C	^L	FF (form feed, new page) <ff>
13.	015	0D	^M	CR (carriage return) <cr>
14.	016	0E	^N	SO (shift out)
15.	017	0F	^O	SI (shift in)
16.	020	10	^P	DLE (data link escape)
17.	021	11	^Q	DC1 (device control 1, XON)
18.	022	12	^R	DC2 (device control 2)
19.	023	13	^S	DC3 (device control 3, XOFF)
20.	024	14	^T	DC4 (device control 4)
21.	025	15	^U	NAK (negative acknowledge)
22.	026	16	^V	SYN (synchronous idle)
23.	027	17	^W	ETB (end transmission block)
24.	030	18	^X	CAN (cancel)
25.	031	19	^Y	EM (end of medium)
26.	032	1A	^Z	SUB (substitute)
27.	033	1B	^[	ESCAPE (alter mode, SEL) <esc>
28.	034	1C	^\	FS (file separator)
29.	035	1D	^]	GS (group separator)
30.	036	1E	^^	RS (record separator)
31.	037	1F	^_	US (unit separator), EOL on some sys
32.	040	20	"_"	space or blank <sp>
33.	041	21	!	exclamation mark
34.	042	22	"	double quote
35.	043	23	#	number sign (hash mark)
36.	044	24	$	dollar sign
37.	045	25	%	percent sign
38.	046	26	&	ampersand sign
39.	047	27	'	single quote (apostrophe)
40.	050	28	(	left parenthesis
41.	051	29	)	right parenthesis

(C) copyright 1981 Alcor Systems

42.	052	2A	*	asterisk (star)
43.	053	2B	+	plus sign
44.	054	2C	,	comma
45.	055	2D	-	minus sign (dash)
46.	056	2E	.	period (decimal point)
47.	057	2F	/	(right) slash
48.	060	30	0	numeral zero
49.	061	31	1	numeral one
50.	062	32	2	numeral two
51.	063	33	3	numeral three
52.	064	34	4	numeral four
53.	065	35	5	numeral five
54.	066	36	6	numeral six
55.	067	37	7	numeral seven
56.	070	38	8	numeral eight
57.	071	39	9	numeral nine
58.	072	3A	:	colon
59.	073	3B	;	semi-colon
60.	074	3C	<	less-than sign
61.	075	3D	=	equal sign
62.	076	3E	>	greater-than sign
63.	077	3F	?	question mark
64.	100	40	@	atsign
65.	101	41	A	upper-case letter ABLE
66.	102	42	B	upper-case letter BAKER
67.	103	43	C	upper-case letter CHARLIE
68.	104	44	D	upper-case letter DELTA
69.	105	45	E	upper-case letter ECHO
70.	106	46	F	upper-case letter FOXTROT
71.	107	47	G	upper-case letter GOLF
72.	110	48	H	upper-case letter HOTEL
73.	111	49	I	upper-case letter INDIA
74.	112	4A	J	upper-case letter JERICHO
75.	113	4B	K	upper-case letter KAPPA
76.	114	4C	L	upper-case letter LIMA
77.	115	4D	M	upper-case letter MIKE
78.	116	4E	N	upper-case letter NOVEMBER
79.	117	4F	O	upper-case letter OSCAR
80.	120	50	P	upper-case letter PAPPA
81.	121	51	Q	upper-case letter QUEBEC
82.	122	52	R	upper-case letter ROMEO
83.	123	53	S	upper-case letter SIERRA
84.	124	54	T	upper-case letter TANGO
85.	125	55	U	upper-case letter UNICORN
86.	126	56	V	upper-case letter VICTOR
87.	127	57	W	upper-case letter WHISKY
88.	130	58	X	upper-case letter XRAY
89.	131	59	Y	upper-case letter YANKEE
90.	132	5A	Z	upper-case letter ZEBRA

(C) copyright 1981 Alcor Systems

91.	133	5B	[	left square bracket
92.	134	5C	\	left slash (backslash)
93.	135	5D	]	right square bracket
94.	136	5E	^	uparrow (carat)
95.	137	5F		underscore
96.	140	60	`	(single) back quote
97.	141	61	a	lower-case letter able
98.	142	62	b	lower-case letter baker
99.	143	63	c	lower-case letter charlie
100.	144	64	d	lower-case letter delta
101.	145	65	e	lower-case letter echo
102.	146	66	f	lower-case letter foxtrot
103.	147	67	g	lower-case letter golf
104.	150	68	h	lower-case letter hotel
105.	151	69	i	lower-case letter india
106.	152	6A	j	lower-case letter jericho
107.	153	6B	k	lower-case letter kappa
108.	154	6C	l	lower-case letter lima
109.	155	6D	m	lower-case letter mike
110.	156	6E	n	lower-case letter november
111.	157	6F	o	lower-case letter oscar
112.	160	70	p	lower-case letter pappa
113.	161	71	q	lower-case letter quebec
114.	162	72	r	lower-case letter romeo
115.	163	73	s	lower-case letter sierra
116.	164	74	t	lower-case letter tango
117.	165	75	u	lower-case letter unicorn
118.	166	76	v	lower-case letter victor
119.	167	77	w	lower-case letter whisky
120.	170	78	x	lower-case letter xray
121.	171	79	y	lower-case letter yankee
122.	172	7A	z	lower-case letter zebra
123.	173	7B	{	left curly brace
124.	174	7C	|	vertical bar
125.	175	7D	}	right curly brace
126.	176	7E	~	tilde
127.	177	7F	<rubout>	DEL <del>

(C) copyright 1981 Alcor Systems

D. Differences from Standard

The standard used is defined by "User Manual and Report", second edition, Jensen and Wirth, Springer-Verlag. The following sections pertain to the differences in Alcor Systems implementation of Pascal as compared to the standard. The extensions are added to provide extra features for programs which are not expected to be transferred to other machines which do not have Alcor Pascal. If transportability is desired, it is advised not to include any of the below listed extensions in the program.

D.1 Omissions

1) The procedures GET and PUT along with associated file buffer variables are not implemented.

2) Procedures or functions may not be passed as parameters to other procedures or functions.

D.2 Extensions

1) Common variables which provide a mechanism for statically allocating local variables are implemented through the use two new declaration parts: COMMON and ACCESS.

2) The declaration sections LABEL, CONST, TYPE, VAR, COMMON, and ACCESS may appear any number of times and in any order within a block.

3) The Type Transfer Operator allows variables to be referenced through the use of a type template.

4) Single elements of packed structures may be passed as parameters.

5) The OTHERWISE clause is implemented in the CASE statement. If omitted, and there is no match, execution transfers to the next statement.

6) Identifiers can include the characters '_' and '$'. Also, no distinction is made between upper and lower case letters.

7) Integer constants or characters may be represented in hex.

8) Mixed mode arithmetic is implemented.

9) The procedures READ or READLN will accept string and boolean variables.

10) External procedures or functions may be declared. This featu provides a way of accessing external routines.

11) Input files are not opened until necessary. This eliminates the synchronization problem when doing interactive input from a terminal.

12) Labels may range from -32768 to 32767.

13) Alternate symbols are implemented for brackets and the pointer symbol.

14) The LOCATION function allows the determination of the address of a variable.

15) The SIZE function allows the size of a type to be determined.

16) The HB function returns the high byte of an integer variable.

17) The LB function returns the low byte of an integer variable.

18) The procedure MESSAGE provides an additional method for handling string output to a terminal.

19) The procedure CLOSE allows components of structured variables to be declared as files.

20) Double precision reals are implemented through the use of a compiler switch option.

21) FOR Loop counter variables are automatically declared if a declaration for them is not present. A compiler switch option may be used to force temporary variables to be used as counters in all FOR statements.

22) Statements can be conditionally compiled through the use of a compiler switch option. This provides an easy way to turn debugging statements on and off.

23) Procedures and functions may be compiled separately through the use of a compiler switch option.

24) Integer values may be output in hexidecimal or decimal base format.

D.2 Other Implementation Characteristics

The following is a list of specific implementation decisions which are not defined by the standard.

1) Only the first 8 characters of an identifier are stored. This means that identifier names should be selected such that the first 8 characters form a unique name.

2) There is a limit of 256 elements for sets, enumerations, CASE statement labels, and parameters to a procedure or function.

3) Pascal source is restricted to 72 columns. This allows room for line numbers when listings are directed at an 80 column terminal or printer.

The following is a list of characteristics which are slightly altered from the standard.

1) Operator precedence has been altered to eliminate the need for excessive use of parentheses in expressions. The precedence is the same as that used in Basic. The difference is the precedence assigned to the Boolean operators. The precedence defined by the standard makes the Boolean operator OR equal in precedence with + and -, the Boolean operator AND equal in precedence with *, /, DIV, and MOD, and NOT has the highest precedence of any operator except the parentheses. Parentheses may be used when transportable programs are being written to maintain compatability with the standard.

2) The PROGRAM statement is not used to associate logical file names to physical files or devices. The association is made either interactively from a terminal or through a procedure call.

3) A GOTO statement may not reference a label outside the block in which the statement appears.

(C) copyright 1981 Alcor Systems

E. THE TYPE STRING

The standard Pascal string is defined to be a PACKED ARRAY OF CHAR. Variables of this type are restricted to a predetermined size. (ie. the size of the array must be specified and cannot be altered during program execution). The predefined type STRING is dynamic. The size of a variable declared as type STRING is determined during program execution. Variables of this type may change in size as the program executes. In addition, variables of type STRING may be used in conjunction with a runtime library of string manipulation functions.

Syntax of type STRING:

```
---> STRING --->
```

Example:

```
VAR  str1, str2, str3 : STRING;
```

Assigning values to dynamic string variables

A dynamic string may be created through the use of the predeclared transfer function BLDSTR. This function has one parameter which may be either a variable of the type PACKED ARRAY OF CHAR or a string constant. The function returns a dynamic string of the same length as the array or string constant passed to it.

Example:

```
str1 := BLDSTR('literal string constant');

str2 := BLDSTR(stringconstant);

str3 := BLDSTR(arrayvariable);
```

(C) copyright 1981 Alcor Systems

The procedures READ and READLN have been extended to accept variables of the type STRING. When a variable of type STRING is specified, all characters from the current file pointer to the end of line mark are read. The size of the string is then equal to the number of characters read. If a read is performed while at the end of line mark, the string variable is assigned an empty string. An empty string is a string of zero length.

Example:

```
READ(str1);

READLN(filename,str2);
```

A string variable may be assigned to another string variable. An assignment between two string variables results in both string variables referencing the same string. (ie. both string variables point to the same location in memory)

Example:

```
str1 := str2;
```

NOTE: For most applications, the preferred method of assignment between two string variables is through the use of the library function CPYSTRING.

A string variable may be assigned a string formed by one of the string manipulation functions in the runtime library. For example, there is a function provided which may be used for assignment between two string variables. The function CPYSTR takes a string variable as a parameter and copys it to another location. The string appearing on the left side of the equal sign then references the new location. In other words, instead of having one copy of the string as in the above example, there are now two copies.

Example:

```
str1 := CPYSTR(str2);
```

(C) copyright 1981 Alcor Systems

Outputing dynamic string variables

The WRITE and WRITELN procedures have been extended to accept variables of the type STRING. When a dynamic string is output, the number of characters written is equal to the length of the string.

Converting a dynamic string into an array

Dynamic strings can only be accessed as a whole. (ie. the individual characters of the string cannot be accessed) The predeclared procedure GETSTR will copy a dynamic string variable into a variable of the type PACKED ARRAY OF CHAR. It accepts two parameters. The first parameter is the dynamic string variable. The second parameter is the array variable. The string is left justified in the array. If the string is longer than the array, then it is truncated. If the string is shorter than the array, then the array is padded with blanks.

Example:

```
        GETSTR(strl, arrayvariable);
```

Recovering memory used by a dynamic string

The memory used by a dynamic string may be recovered through the use of the standard procedure DISPOSE. When a string variable is passed to the DISPOSE procedure, the memory used by the string is freed and the string variable becomes undefined. In addition, any other string variable which points to the same string will become undefined. Each time a string variable is assigned a value, it points to a new string and the old string is then lost. The memory it uses cannot be recovered. Therefore, before assigning the string variable a new value, the memory used by the old value should be recovered if space is important.

Example:

```
         strl := BLDSTR('this is the first value');
         ...
         DISPOSE(strl);
         strl := BLDSTR('this is the second');
```

Using the string library

There is a long list of string manipulation functions available in the runtime library. In order for a program to have access to these functions, it must include an external declaration for each function used. A file of external declarations for all the string functions is supplied on disk. The text editor may be used to insert this file into the programs that use these functions. The declarations for any functions which are not used may be deleted if desired. If only one or two functions are used, you may prefer just to type in the external declaration.
(See the System Implementation Manual for a description of the string manipulation functions)

Example use of dynamic strings:

```
PROGRAM sample;

FUNCTION CONC(s1,s2 : STRING) : STRING; EXTERNAL;
(*conc is a library function which concatenates two strings*)


VAR  firstname, lastname,
     space, fullname          : STRING;

BEGIN
  space := BLDSTR(' ');
  WRITELN(' enter first name');
  READLN(firstname);
  WRITELN(' enter last name');
  READLN(lastname);
  fullname := CONC(CONC(firstname,space),lastname);
  WRITELN(fullname);
END;
```

MASTER CROSS REFERENCE INDEX

B = Beginners Guide   E = Editor Manual   R = Reference Manual
S = System Manual   T = Tutorial

MASTER CROSS REFERENCE INDEX

MASTER CROSS REFERENCE INDEX

MASTER CROSS REFERENCE INDEX

MASTER CROSS REFERENCE INDEX

MASTER CROSS REFERENCE INDEX

MASTER CROSS REFERENCE INDEX

[redacted] [redacted]

```
ALCOR PASCAL   VERSION: 1.2A   13000348   00:53:02 00/00/00     PAGE   1
     |------------------------------------------------------------------------|
0000|PROGRAM BUILDFILE(DATA725,OUTPUT);
0000|
0000|
0000|   VAR
0000|      DATA725:TEXT;
0000|
0000|         BEGIN
0000|            REWRITE(DATA725);
0047|                 WRITE(DATA725,'36,100,21,58,33,100,28,74');
006C|
006C|        END.
NO ERRORS DETECTED
```

BUILD 725/PCL

```
ALCOR PASCAL   VERSION: 1.2A   13000348   00:54:37 00/00/00     PAGE   1
     |------------------------------------------------------------------------|
0000|PROGRAM LOAD1(OUTPUT,DATA725);
0000|   VAR
0000|      DATA725:TEXT;
0000|      HEADS,FEET:INTEGER;
0000|
0000|   BEGIN
0000|      RESET(DATA725);
0047|      WHILE NOT EOF(DATA725)DO
004F|         BEGIN
004F|            READ (DATA725,HEADS,FEET);
0061|               WRITELN(HEADS,FEET);
007D|         END;
007F|   END.
NO ERRORS DETECTED
```

READ1/PCL

```
      36      100
      21       58
      33      100
      28       74
```